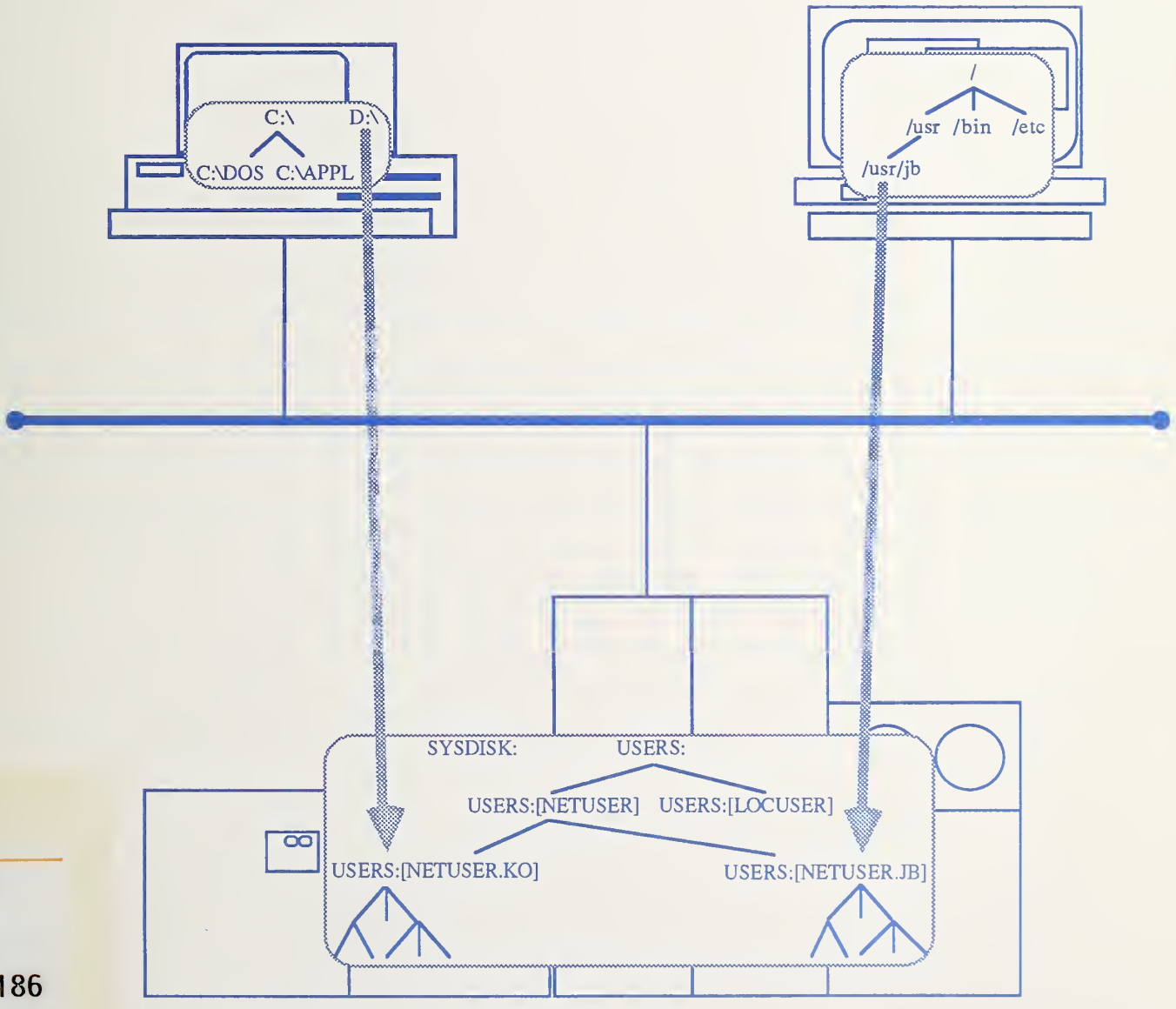
NIST Special Publication 500-186

Computer Systems Technology

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

Issues in Transparent File Access

Karen Olsen
John Barkley



NIST Special Publication 500-186

Issues in Transparent File Access

Karen Olsen
John Barkley

Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

April 1991

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
John W. Lyons, Director

Reports on Computer Systems Technology

The National Institute of Standards and Technology (NIST) has a unique responsibility for computer systems technology within the Federal government. NIST's Computer Systems Laboratory (CSL) develops standards and guidelines, provides technical assistance, and conducts research for computers and related telecommunications systems to achieve more effective utilization of Federal information technology resources. CSL's responsibilities include development of technical, management, physical, and administrative standards and guidelines for the cost-effective security and privacy of sensitive unclassified information processed in Federal computers. CSL assists agencies in developing security plans and in improving computer security awareness training. This Special Publication 500 series reports CSL research and guidelines to Federal agencies as well as to organizations in industry, government, and academia.

National Institute of Standards and Technology Special Publication 500-186
Natl. Inst. Stand. Technol. Spec. Publ. 500-186, 86 pages (Apr. 1991)
CODEN: NSPUE2

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1991

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402

EXECUTIVE SUMMARY

The use of computer systems, such as workstations, personal computers, minicomputers, and mainframes, in a network environment is pervasive. The Open Systems concept is a major factor in the evolution of these systems. For a system attached to a network, the network provides connectivity to many other systems whose file systems may be very different from the local file system, e.g., the local may be hierarchical and the remote flat. An application has potential access to information located on file systems beyond the local file system. However, there is no standard way for that application to be able to *transparently* access files on several file systems whose access characteristics may differ from the access characteristics of the local file system. Transparent file access means that remote files are accessed as though they were local.

The Institute of Electrical and Electronics Engineers (IEEE) 1003.8 Transparent File Access (TFA) Working Group of the POSIX Standards Committee (IEEE P1003) has undertaken the development of an application programming interface specification based on the IEEE 1003.1-1990 Standard which:

- Provides a standard means of characterizing and profiling file systems.
- Permits access to the widest possible range of file systems which can resemble the file system of IEEE 1003.1-1990.
- Provides a means for an application program to simultaneously manipulate files whose access characteristics differ.

The IEEE 1003.8 TFA Standard under development is suitable for use by any application, regardless of the application's complexity or the complexity of the file system which the application accesses. For a simple application, such as a word processor, the IEEE 1003.8 TFA Standard provides an access specification for rudimentary file systems, e.g., a flat file system. At the same time, the IEEE 1003.8 TFA Standard provides an access specification for robust file systems, e.g., with read/write consistency and record locking, capable of supporting complex applications, such as a database management system.

This report provides the background needed to understand the IEEE 1003.8 TFA Standard whose development is ongoing as this report is published. This report presents the major issues and problems whose resolution forms the basis of the IEEE 1003.8 TFA Standard. These issues are categorized as:

1. Semantic Issues which arise as a result of attempting to implement some aspects of IEEE 1003.1-1990 in an environment for which it was not designed, e.g., a network environment.
2. Performance Issues which arise as a result of the performance penalty that may result from attempting to implement some aspects of IEEE 1003.1-1990 in an environment for which it was not designed, e.g., a network environment.

3. Specific Environment Issues which arise because of the need for IEEE 1003.8 TFA to be applicable to specific or emerging file systems or network protocols.

4. Miscellaneous Issues.

Some issues are illustrated with examples and demonstrations using NFS, the most widely used implementation for accessing remote files on a network. The issues chosen for demonstration are based on their importance, complexity, and ease of reproducibility.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The low cost of computer hardware has made the personal computer and the workstation an integral part of the workplace. As a result, the need for communications between these systems has increased dramatically. Not only has the quantitative need increased, but the nature of that need has changed. Previously, the ability to log into remote systems and to transfer files was sufficient for users of personal computers and workstations. Now it is becoming the norm to access the services available through a communication network in a manner consistent with accessing the services of the local system.

Such "transparent" services include electronic mail and the use of remote devices, e.g., disks, tapes, and printers, as though they were physically local. Previously, access to electronic mail usually required the user to log into a remote large system in order to send and receive mail. Now, it is common for a user to run software on the local system which transparently accesses a remote mail server. The user's mail is actually sent and received by the mail server. However, the user perceives that the mail is sent and received by the local system which, in many cases, is too small to act as a mail server.

Previously, access to remote devices, such as disks and printers, was usually provided only by first transferring files to the system where the device was physically connected. Now, access to remote devices is provided as though the devices were connected locally. In the case of accessing remote file systems, such access is referred to as "transparent file access." A user need not know whether a device is local or remote.

Application programs are the means by which services are provided to users. Application programs that make use of transparent file access require an application programming interface and the support of network protocols. Network protocols are required in order to accomplish the communication between systems. This communication is necessary so that remote file systems appear local. The application program need not be directly concerned with network protocols. The network protocols are provided as a result of the application program's use of the application programming interface. The application programming interface reflects the semantics of the file system being used by the application program.

For example, the Network File System (NFS)[1] developed by Sun Microsystems is a protocol which is layered above the TCP/IP (Transmission Control Protocol/Internet) protocol suite. The application programming interface for NFS is the Unix application programming interface which provides most of the the file system semantics of Unix. Another example is NETBIOS which provides the DOS Redirector network access using the SMB protocol. Application programs use the DOS application programming interface which provides the file system semantics of DOS over a network by means of the DOS Redirector.

Both standard network protocols and a standard application programming interface are necessary for transparent file access in order to ensure interoperability between systems and portability of applications. The transparent file access environment can be described by means of a client/server model (see cover illustration). A client, such as the personal computer in the cover illustration, accesses both local and remote files. Local files, such as those on C: in the cover illustration, are resident on a disk which is directly connected to the client. Remote files, such as those on D: in the cover illustration, are physically located on a server which is connected to a network as is the client. The application program need not be aware of whether a file is local or remote.

Regardless of the physical location of a file, the client accesses the file according to the file system semantics of the client's operating system. The file system semantics of the client operating system may differ from the file system semantics of the server. Moreover, the file system semantics of two clients accessing the same server may be different from each other and different from the server. It is a function of the client implementation and the server implementation as to who, i.e., the client, the server, or both, has the responsibility for guaranteeing the client's file system semantics when server files are accessed.

For example, as illustrated on the cover, a personal computer using DOS and a workstation using Unix may be accessing files from a VAX VMS server. The personal computer, the workstation, and the VAX must be using the same network protocols in order to interoperate. Consider application programs running on the personal computer, the workstation, and the VAX. The application program on the personal computer must access files using the DOS file system semantics; the application program on the workstation must access files using Unix file system semantics; and the application program on the VAX must access files using VMS file system semantics. Such an environment is not conducive to portable applications. A standard application programming interface is necessary in order for application programs to be portable among clients and servers in environments such as these.

Under development in the IEEE Transparent File Access (TFA) Working Group, 1003.8 is a revision to IEEE 1003.1-1990 (ISO/IEC 9945-1:1990). The TFA Working Group is part of the IEEE POSIX Standards Committee (IEEE P1003). This revision provides a file system access specification which would permit access to the widest possible range of file systems which can resemble the file system of IEEE 1003.1-1990. The Transparent File

[1]Because of the nature of this report, it is necessary to mention vendors and commercial products. The presence or absence of a particular trade name product does not imply criticism or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available.

Access Standard of IEEE 1003.8 provides the means for an application to simultaneously manipulate files whose access characteristics differ because they belong to file systems with different file system semantics. In addition, the specification for the first time provides a standard way of characterizing and profiling the capabilities of file systems.

This report provides the background needed to understand the TFA Standard. Because the development of the TFA Standard is ongoing as this report is published, this report does not reference any Draft of the TFA Standard and only presents the significant issues which are motivating the development of the TFA Standard. The report goes beyond the rationale of the TFA Standard, which summarizes why certain choices were made by the TFA Working Group in their development of the TFA Standard. This report presents an in-depth description of the major issues on which the choices were based.

The report assumes that the reader has some knowledge of Unix. Since the issues described are common to most Unix implementations, the issues are presented in such a manner as to be understandable to a reader who is familiar with some Unix implementation. Although all issues presented are issues for IEEE 1003.1-1990, familiarity with IEEE 1003.1-1990 is not required. Where IEEE 1003.1-1990 differs from a traditional Unix implementation with regard to an issue, that difference is mentioned. It is not intended that the description of an issue be a tutorial on the specific Unix functionality involved in the issue. Only those aspects of Unix functionality which directly impact the issue are highlighted.

Some issues are illustrated with examples and demonstrations using the NFS implementation of SunOS 4.1 from Sun Microsystems on both the client and the server. In this report, the term NFS always refers to the NFS Version 2 Protocol. Within the text which describes each demonstration, the term NFS refers to the client and server implementation of the NFS Version 2 Protocol in SunOS 4.1. The results of the demonstrations may be different if other NFS client or server implementations are used. NFS is the most widely used implementation for accessing remote files in a network environment. For those issues chosen for demonstration, the choice was based on the importance and complexity of the issue, and how amenable the issue was to an easily understood and reproducible demonstration.

This report is intended for managers, programmers, and users in government and industry to assist in their understanding, evaluation, management, and use of the TFA Standard and systems which provide access to remote file systems. The main body of the report discusses the issues which arose during the development of the TFA Standard. TFA issues may be characterized as belonging to one of the following categories:

1. Semantic Issues (ch. 2)

   These are issues which arise when trying to implement some aspects of IEEE 1003.1-1990 in an environment for which it was not designed, e.g., a network environment. IEEE 1003.1-1990 applies only to local file systems. Some capabilities cannot be implemented within the context of IEEE 1003.1-1990 in a network environment.

2. Performance Issues (ch. 3)

   These are issues which also arise when applying IEEE 1003.1-1990 to an environment for which it was not specifically developed. However, instead of relating specifically

to capabilities which cannot be implemented within the context of IEEE 1003.1-1990, these issues concern capabilities whose implementation may result in an unacceptable level of performance.

3. Specific Environment Issues (ch. 4)

   These are issues which arise because of the need for the TFA Standard to be applicable to specific existing or emerging file systems or network protocols.

4. Miscellaneous Issues (ch. 5)

   These are issues which do not clearly fit in any of the other categories or which have implications for all of the other categories.

While some issues may have implications for more than one category, each issue is listed only within the category where it has the most significance. Chapter 6 summarizes the results of this report. Appendix A contains references and related reading. Appendix B contains listings of the programs used in the demonstrations. Assistance in the debugging of these program listings by Dan Nielsen and Craig Sparkman is much appreciated.

Chapter 2

Semantic Issues

Semantic issues are those issues which arise when trying to implement some aspects of IEEE 1003.1-1990 in an environment for which it was not designed, e.g., a network environment. IEEE 1003.1-1990 applies only to local file systems. Some capabilities cannot be implemented within the context of IEEE 1003.1-1990 in a network environment. Semantic issues include sending a SIGPIPE signal when using a remote FIFO (see sec. 2.1), guaranteeing that data written by one process will not be interleaved with data written by another process (see sec. 2.2), and returning the value of *l_pid* by *fcntl()* (see sec. 2.3).

2.1 Remote FIFOs and SIGPIPE

When a process writes to a FIFO (see sec. 4.3 for a definition of FIFO) and no process has that FIFO open for reading, the write fails with an EPIPE error condition. In addition, a SIGPIPE signal is sent to the process trying to write to indicate that a write was attempted on a FIFO with no readers.

In a network environment, it is possible that processes using a remote FIFO on a server may not all be located on the same client. In this case, since the server coordinates access to the FIFO, the server notifies the client of the EPIPE error condition and of the necessity of sending the SIGPIPE signal to the process. Thus, this issue is a protocol issue, not an issue requiring a modification to the semantics associated with SIGPIPE. The semantics of SIGPIPE on the client can be maintained as long as the client is notified by the server that the EPIPE error condition has occurred.

2.2 Non-interleaved Writes

When a process writes data to a file, each implementation determines how much data can be written in one atomic operation, i.e., a single operation which cannot be interrupted. For example, a process does a *write(fildes,buf,nbytes)*. The *nbytes* of data may not be written as one atomic operation. The implementation may write the *nbytes* of data in several oper-

ations or the write operation may be interrupted. This situation leaves open the following possibilities:

1. The block of data written by one process is interleaved with data written by another process.

2. Parts of the data written by one process are overwritten by data from other processes before all of the data from the first process has been written.

For many implementations (not including IEEE 1003.1-1990), there is a value of *nbytes* which guarantees that neither of the situations above will occur. In some cases, an application may need to know how much data it can write without being interleaved or overwritten by another process. One example of such an application is a server process which uses a pipe as an input request queue to accept service requests from several other processes.

If the file system is local, it is possible that for all values of *nbytes*, an implementation may write all *nbytes* of data in one operation and that operation is not interruptible. However, if the file system is remote, it is more likely that there is a maximum number of bytes that can be written in one atomic operation. The reason for this is that transparent access to a remote file system may be implemented using existing network protocols. Such protocols may already be standardized and specify a maximum packet size. This maximum packet size may become the maximum number of bytes that can be written atomically. Thus, for a remote file system, the maximum number of bytes that can be written in a single atomic operation is usually the minimum of:

1. the maximum number of bytes which the client implementation can write atomically, and

2. the maximum number of bytes allowed in a packet by the underlying protocols supporting the transparent file access.

One solution to this problem is to specify the maximum number of bytes which may be written in one atomic operation as a parameter which the application program can obtain. A similar approach is already used for pipes. Because of the nature of pipes, it is usually not practical to guarantee that write operations of any size are not interleaved by write operations from other processes. In IEEE 1003.1-1990, the parameter PIPE_BUF specifies the maximum number of bytes which can be written atomically to a pipe or a FIFO. A similar parameter could be used for regular files. Note that in the local environment, the value of PIPE_BUF applies to pipes and FIFOs. However, in a transparent file access environment, the value of a parameter for files similar to PIPE_BUF may be different for each remote file system.

2.3 *l_pid* Returned by *fcntl()*

A process can check to see if a file has a lock on any data in the file by using F_GETLK with *fcntl()*. The *l_pid* field of the *flock* structure may be used under certain conditions with

F_GETLK to return the process ID of the process holding the lock. Existing applications commonly use the *l_pid* returned to kill the process holding the lock, thereby releasing the lock.

In a network environment, processes may be running on different nodes. Consider what would happen if the *l_pid* returned refers to a process on another node. An application might kill an innocent process on its own node. This situation arises because F_GETLK with *fcntl()* is being used in an environment for which it was not designed. This is a semantic issue because the semantics of traditional Unix and IEEE 1003.1-1990 must be modified to avoid this situation.

There are several ways to modify the semantics of *fcntl()* to deal with this problem. One way would be to have a process ID space which would span several systems on a network, i.e., a given *pid* would uniquely identify both the process and the system on which the process runs. Then, a *pid* applied to a *kill()* could correctly signal the intended process. Another approach would be to have a unique process ID value which could refer to a process "not known to the system." When this unique value for *pid* is used as an argument to *kill()* (or any other function which takes *pid* as an argument), the function would return an error.

Demonstrations were developed using an NFS implementation to illustrate for different client/server configurations what happens when a file is locked and F_GETLK with *fcntl()* is used to show the value of *l_pid* returned. Figure 2.1 illustrates how the demonstration proceeds. Commands are displayed in *italics* and the output of those commands is displayed in **bold**. The program *setlock* locks an entire file, waits for a newline, and then unlocks the file. The program *getlock* uses F_GETLK to show the value of *l_pid* returned for different client/server configurations. Source code for the demonstration programs (*setlock.c* and *getlock.c*) is located in Appendix B. For the following demonstrations, Process A is the program *setlock*, Process B is the program *getlock*, and *testfile* is the file which is locked. In addition to Process A and Process B, another process on the system running Process A uses the "*ps*" command to confirm that the value for *l_pid* returned by Process B is indeed the process ID of Process A. Note that in these demonstrations, the Status Monitor and the Lock Manager must be running on both the client and the server.

Table 2.1 illustrates the use of Processes A and B in four different client/server configurations as follows:

- Process A and Process B are on the same client.

  Process A and Process B are on the same client and *testfile* is remote to both processes. Process A locks *testfile* and Process B gets the process ID of the process holding the lock on *testfile*. In this case, the *l_pid* returned is indeed the process ID of Process A, the process holding the lock on *testfile*. Notice that for the NFS implementation, the correct *l_pid* is returned even though the file is remote.

- Process A is on the client and Process B is on the server.

  The file *testfile* is remote to Process A and local to Process B. Process A on the client locks *testfile*. Process B on the server gets the process ID of the process holding the

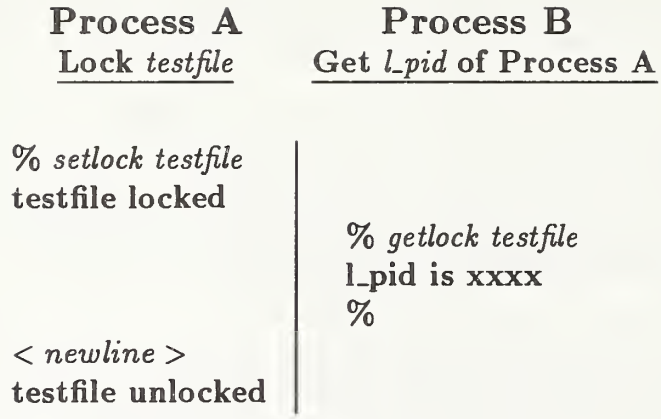


Figure 2.1: *l_pid* returned by *fcntl()* demonstration procedure.

lock on *testfile*. The *l_pid* returned to Process B on the server is the correct process ID of Process A on the client. Even though the correct *l_pid* is returned on the server, the process ID does not refer to a process on the server node. Traditional Unix semantics do not enable a process to be aware that the *l_pid* returned in this case is on another network node.

- Process A is on the server and Process B is on the client.

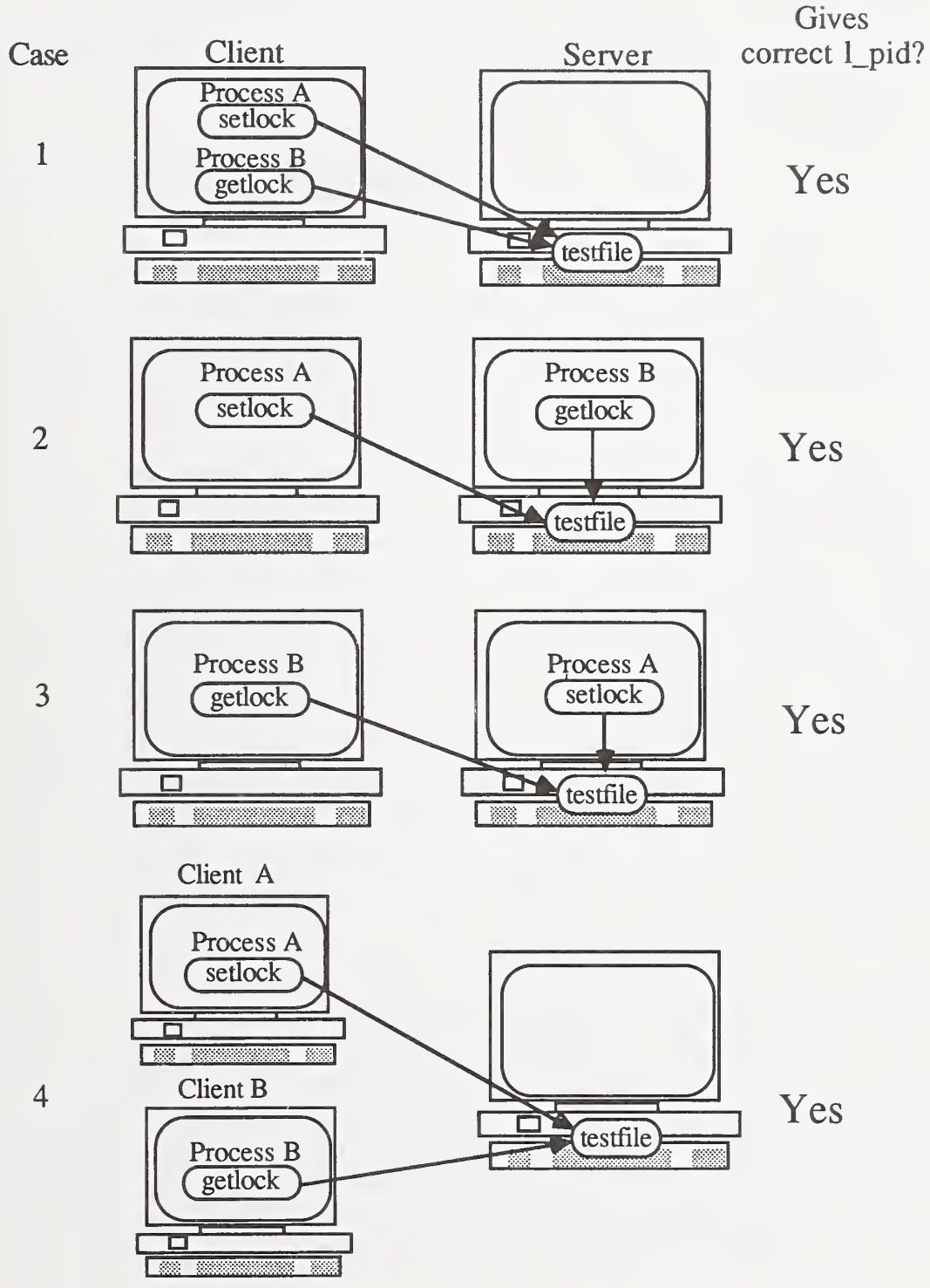
  The file *testfile* is local to Process A and remote to Process B. Process A on the server locks *testfile* and Process B on the client obtains the *l_pid* of the remote process holding the lock on *testfile*. The correct *l_pid* for Process A is returned but as in the previous case, Process B may not be aware that Process A is a process on another system.

- Process A is on Client A and Process B is on Client B.

  The file *testfile* is remote to both Process A and Process B. Process A on Client A locks *testfile*. Process B on Client B gets the *l_pid* of the remote process holding the lock on *testfile*. The correct *l_pid* is returned. As in the previous two cases, the *l_pid* identifies Process A but does not identify the system on which Process A is running.

The NFS implementation makes F_GETLK and *l_pid* work correctly in all cases. This is accomplished by two additional protocols, the Lock Manager protocol and the Status Monitor protocol, used in conjunction with NFS. The Status Monitor protocol permits both clients and servers to be aware of what systems are up and running. The Status Monitor informs interested applications when a system failure occurs on a system which it is monitoring. The Lock Manager protocol implements record locking on NFS mounted file systems. A process on a remote system locking a file is identified by the Lock Manager and that value is returned by *fcntl()*. However, *fcntl()* does not identify the system on which the process holding the lock is running.

Table 2.1: *l_pid summary*



Chapter 3

Performance Issues

Performance issues are those issues which arise when applying IEEE 1003.1-1990 to a network environment, an environment for which the standard was not specifically designed. Unlike semantic issues, performance issues are IEEE 1003.1-1990 facilities that can be implemented in a network environment, but may result in performance degradation.

Caching is one of the most widely used techniques for improving the performance of data transfers between a system and an external device. The technique may be applied to many different situations. Applying this technique to system and application software means that data transfers initiated by programs may take place between a program's local memory and a *cache* memory instead of directly to the external device. Small data transfers initiated by a program are combined into large blocks in the cache memory. Data transfer of the large blocks between the cache memory and the external device is managed by the system. The use of caching permits data transfers to or from the external device to take place in larger blocks in order to better overlap CPU operation and external device access, thus improving system throughput. In addition, since data is temporarily stored in large blocks in cache memory, the number of accesses to the external device is reduced. The data may already be available in the cache for reading and each write of a small block of data goes to the cache. Caching data transfers between system and application software may be useful any time access is required to an external device whose speed of access differs significantly from that of memory. For example, caching is typically used to improve the performance of disk access and network access by both clients and servers. When accessing remote files in a network environment, client caching is used to write and read data from cache on the client instead of directly writing to and reading from the network. Using client caching improves client system performance as well as decreases the number of network accesses.

Client caching plays such an important role in improving performance and decreasing network access that most clients use some caching mechanism when accessing remote files. A demonstration was developed using NFS to show the effects on performance by forgoing client caching. The program *read_file* (see Appendix B) reads a file one byte at a time and displays the byte. For this demonstration, *testfile* is the remote file being read. The effects of using client caching and turning off client caching are summarized below.

- Client caching.

  The program *read_file* is run from a client which is using client caching. Data from the server is read into the client's cache. The program *read_file* reads *testfile* one byte at a time from cache. When all the data in cache has been read, cache is refilled with data from the server. The network is accessed only when the client cache needs to be filled. The file *testfile* is displayed at a speed that is approximately the same as if *testfile* were local.

- Turning off client caching.

  The program *read_file* is run from a client which has client caching turned off for the remote file *testfile*. For this demonstration, client caching is turned off by running *setlock testfile* as a separate process on the same client. The program *setlock* applies an advisory lock to all bytes of *testfile*. This turns off caching for *testfile*. The lock on *testfile* is initiated before *read_file* begins and is maintained until *read_file* completes. Because client caching is turned off, each byte of data is read directly from the server. This means that the client must access the network each time a byte of data is read. The file *testfile* is displayed at a speed much slower than if *testfile* were local. If *testfile* is a large file, the decline in performance is even more pronounced than if *testfile* were small.

This demonstration shows that the level of performance resulting from not using client caching is often unacceptable. Not only may a severe performance degradation result, but the network is bombarded with packets of data being sent, one byte at a time, between the client and the server.

This section deals with IEEE 1003.1-1990 facilities that can be implemented in a network environment, but can incur a performance penalty. Section 3.1 contains demonstrations which illustrate read/write consistency behavior for different client/server configurations. The effects of using Unix IO, *stdio*, and client caching, are summarized. Section 3.2 discusses the problems associated with the times returned by *stat()*. This issue is a semantic issues as well as a performance issue. The reporting of *write()* error conditions is discussed in section 3.3. Section 3.4 discusses record locking.

3.1 Read/Write Consistency

Read/write consistency, which means that data written by a process becomes "visible" immediately after a *write* returns, is explicitly guaranteed by IEEE 1003.1-1990. Figure 3.1 illustrates the concept of read/write consistency for two processes, Process A and Process B, which are running on the same node using Unix IO, not *stdio*, and are accessing local data. In this section, the term "Unix IO" is used to refer to file access which does not use *stdio*. When Process A or Process B executes a *write()*, the data goes to a cache for the device. When either Process A or Process B executes a *read()*, the data is read from cache. Read/write consistency is maintained because both processes read directly from the same

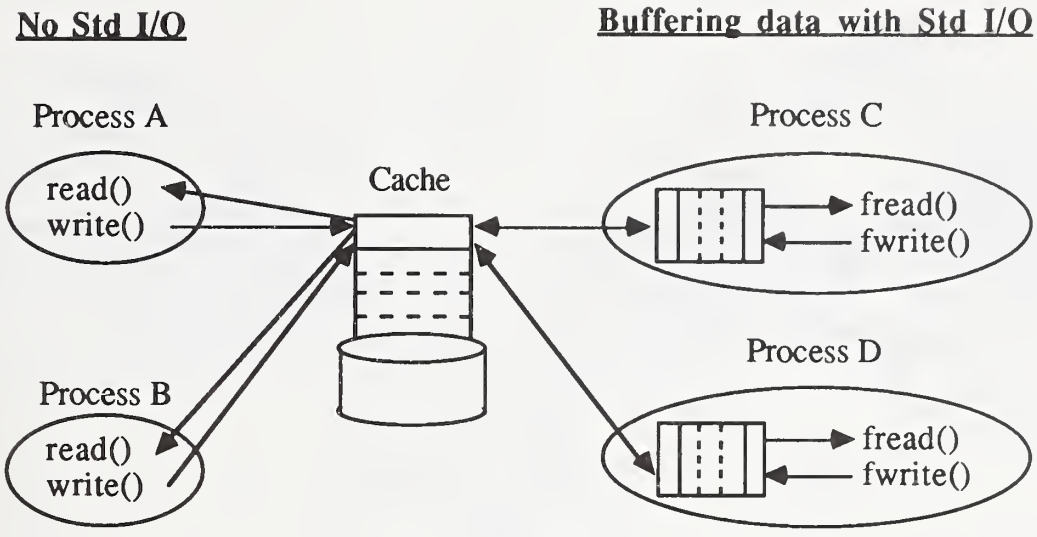


Figure 3.1: Local case for *read()* and *write()*.

cache. When the *write()* of Process A or Process B returns, the data is "visible" immediately to any other process on the system using Unix IO. There is a demonstration of this at the end of this section (see fig. 3.4). If caching is not used, data goes directly to the device.

There are several situations which involve the concept of read/write consistency when accessing local files. These are:

1. Single Process.

   A process opens a file, writes some data at the beginning of the file, uses *lseek()* to position the file pointer to the beginning of the file, and reads. This case is read/write consistent. The process reads what it just wrote.

2. Processes sharing an open file description.

   A process opens a file, writes some data at the beginning of the file, forks a child who uses *lseek()* to position the file pointer to the beginning of the file and reads. This case is read/write consistent. The child process reads what the parent process just wrote.

3. Processes not sharing an open file description (see Process A and Process B in fig. 3.1).

   Process A opens a file and writes some data at the beginning of the file. Some time after the write of Process A returns, Process B opens the file, positioning the file pointer to the beginning of the file, and reads. This case is read/write consistent. Process B reads what Process A just wrote.

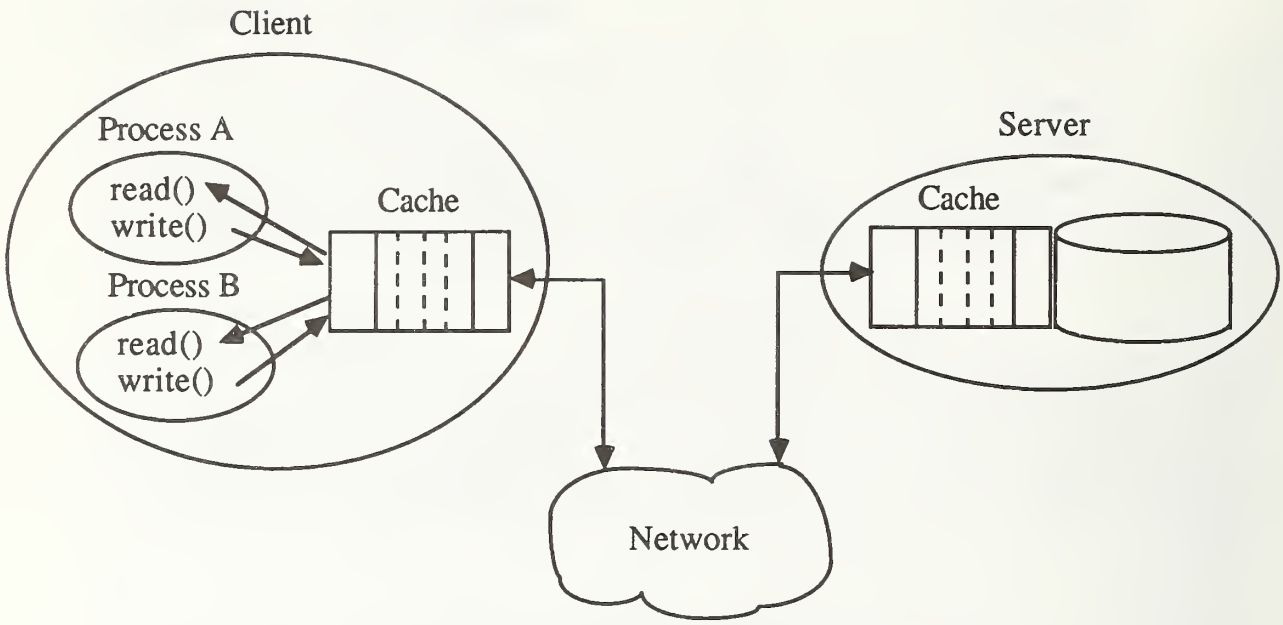


Figure 3.2: *read()* and *write()* for two processes on the same client.

4. *Stdio.*

   Figure 3.1 shows two processes, Process C and Process D, which use *stdio* to buffer data for accessing local files. In order to improve performance, data written using *stdio* usually goes to a buffer associated with a process. The data is not visible to any other process until the buffer is flushed. When an *fwrite()* returns, the data may have been written to disk or the data may still be in the buffer. An initial *fread()* will read data from the disk into a buffer associated with the process. Subsequent *fread()s* will read data from the buffer. Once all the data in the buffer has been read, the buffer is refilled with data from disk. It is possible for data in the buffer to be inconsistent with data on disk. Therefore, read/write consistency fails when using *stdio*. There is a demonstration of this case at the end of this section (see fig. 3.5).

Read/write consistency is also an issue in a network environment. In order to improve performance, most client implementations use some caching mechanism when accessing remote files. Processes on the same node are usually read/write consistent because they read from and write to the same cache (see fig. 3.2). However, processes on different nodes using client caching may not be read/write consistent because the processes read from and write to different caches.

The lack of read/write consistency when using *stdio* on the same node is analogous to the lack of read/write consistency when processes are running on different nodes. Both *stdio* and client caching are used to improve performance. With *stdio*, each process has a set of read and write buffers. Thus, read/write consistency is usually only maintained at the level of a

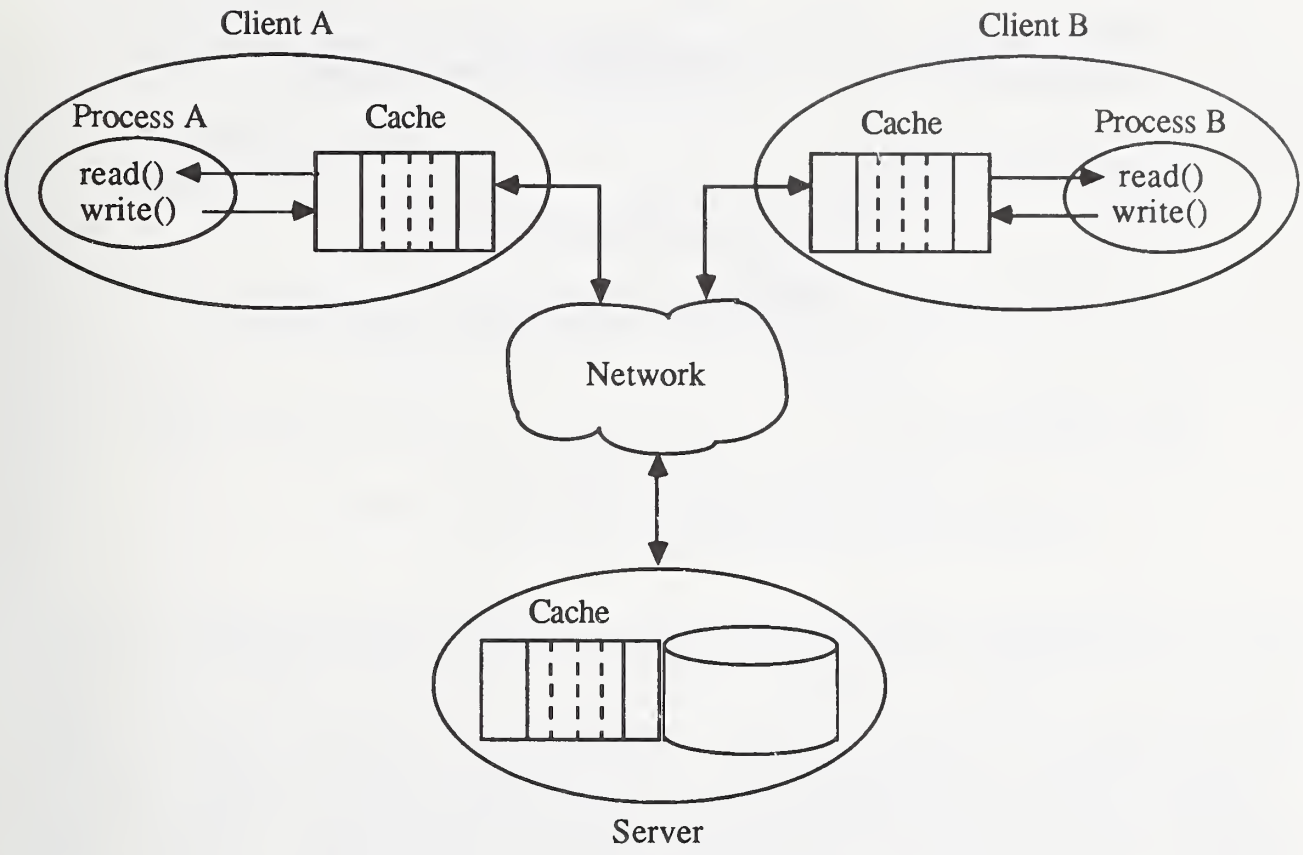


Figure 3.3: *read()* and *write()* for two processes on different clients using client caching.

single process. With client caching in a network environment, read/write consistency may only be maintained for the set of processes on a single client. Processes on different clients may not be read/write consistent when client caching is used. One way to ensure read/write consistency among processes on different nodes is to forgo the use of client caching. This means that when Unix IO is used, all writes are sent to the server before the *write()* returns, and all reads are obtained directly from the server before the *read()* returns. This usually entails a performance penalty. There are techniques which permit the use of client caching and still maintain read/write consistency. Appendix B contains references for some of these techniques.

Figure 3.3 depicts two processes, Process A and Process B, which are running on different clients and use client caching. Assume that both processes are accessing the same file on the server. An initial *read()* by a process will read data from the disk on the server into the cache associated with each client. Subsequent *read()s* by a process will read data from client cache. If Process A executes a *write()*, the data may not be written immediately to the server. All writes on the client are cached and are written to the server at some later time. Therefore, it is possible for data in cache to be inconsistent with the server's data. In

Process A *WriteUnixIO*	**Process B** **Display file contents and size**
% *WriteUnixIO outfile* **abcde**	
	% *cat outfile; echo " "; ls -l outfile* **abcde** **-rw-r--r-- 1 olsen 5 Jul 19 15:26 outfile** %
^D **read/wrote 5 bytes**	

Figure 3.4: A demonstration of read/write consistency.

order for Process B to see the data that Process A just wrote, Client A would have to write the data in its cache to the server, and Client B would have to fill its cache with new data from the server. Processes on different nodes may not be read/write consistent when using client caching because the data may not be "visible" to all processes after a *write()* returns.

Demonstrations were developed using an NFS implementation to illustrate read/write consistency. The following demonstrations use the programs *WriteUnixIO* and *WriteStdIO*. Source code for these programs is located in Appendix B. *WriteUnixIO* is a program which reads input from the terminal and writes output to *outfile*. The file name *outfile*, which in the demonstrations refers to either a local file or a remote file mounted under the directory *mnt*, is a parameter to *WriteUnixIO*. Data is read and written a byte at a time using Unix IO. *WriteStdIO* is similar to *WriteUnixIO* except that the program uses *stdio* to buffer data for writing. Periodically, the command "cat outfile; echo " "; ls -l outfile" is run to display the contents and the size of *outfile*. This command shows whether or not data is "visible" immediately after a *write()* returns. For all the demonstrations, Process A is either the program *WriteUnixIO* or *WriteStdIO* and Process B is a process which displays the contents and size of *outfile*. Figure 3.4 and figure 3.5 illustrate how the demonstration proceeds. Commands are displayed in *italics* and the output of those commands is displayed in **bold**.

- Unix IO on a single system (see fig. 3.4).

  Process A and Process B are both on the same single system and *outfile* is a local file. Process A is the program *WriteUnixIO*. After data is entered, Process B displays the contents of *outfile*. Data is "visible" to Process B immediately after a *write()* from Process A returns. Unix IO on a single system is read/write consistent.

- *Stdio* on single system (see fig. 3.5).

  Process A and Process B are both on the same single system and *outfile* is a local

Process A Run *WriteStdIO*	Process B Display file contents and size
% *WriteStdIO outfile* **abcde**	
	% *cat outfile; echo " "; ls -l outfile* **-rw-r–r– 1 olsen 0 Jul 19 15:26 outfile** %
^D **read/wrote 5 bytes**	
	% *cat outfile; echo " "; ls -l outfile* **abcde** **-rw-r–r– 1 olsen 5 Jul 19 15:26 outfile** %

Figure 3.5: A demonstration of the lack of read/write consistency.

file. Process A is the program *WriteStdIO*. After data is entered, Process B displays the contents of *outfile*. Because *stdio* is used to buffer output, data is not "visible" to Process B immediately after a *write()* from Process A returns. *Stdio* on a single system is not read/write consistent. The data becomes "visible" when a *Ctrl-D* is received which causes the buffer to be dumped to disk.

- NFS with processes on the same node (see fig. 3.4).

  Process A and Process B are on the same client and *outfile* is a remote file. Process A is the program *WriteUnixIO*. After data is entered, Process B displays the contents of *outfile*. Data is "visible" to Process B immediately after a *write()* from Process A returns. Two processes on the same node using NFS are read/write consistent.

- NFS with processes on different nodes (see fig. 3.5 replacing *WriteStdIO* with *WriteUnixIO*).

  Process A is on Client A, Process B is on Client B, and *outfile* is remote to both processes. Process A is the program *WriteUnixIO*. After data is entered, Process B displays the contents of *outfile*. Because client caching is used, data is not "visible" to Process B for several seconds after a *write()* from Process A returns. Processes on different clients which use client caching are not guaranteed to be read/write consistent. The data becomes "visible" to Process B when Client A's cache is written to the server and Client B's cache is refilled from the server. This is similar to the read/write consistency issue when using *stdio*.

- NFS with processes on different nodes without client caching (see fig. 3.4).

  Process A is on Client A, Process B is on Client B, and *outfile* is remote to both Process A and Process B. Process A is the program *WriteUnixIO*. Before Process A is started, another process on Client A runs *setlock outfile* to lock *outfile* in the same manner as was used in the demonstration on the effect of client caching on performance. Advisory record locking is one method of turning off client caching for individual files. Although Client B uses client caching, each time the command line of Process B is run, *outfile* is opened, read in its entirety, and closed. The important thing to note from this demonstration is that a *write()* from Process A is "visible" immediately to Process B after the *write()* returns because the data is written directly to the server instead of to Client A's cache. Also note that even though all bytes in *outfile* have been exclusively locked, Process A is able to write to *outfile* because the locking mechanism is advisory and not mandatory. Read/write consistency is guaranteed, using NFS, for processes on different nodes that do not use client caching.

Processes on different nodes that do not use client caching are read/write consistent, but processes on different nodes that do use client caching may not be. Applications that need to guarantee read/write consistency should use record locking. In some implementations where client caching is used, record locking is the preferred method of guaranteeing read/write consistency. Because of possible performance degradation from providing read/write consistency all of the time, some implementations only guarantee read/write consistency among processes who use record locking for simultaneous file access.

Table 3.1 summarizes the results of the read/write consistency demonstrations. For those cases that are not read/write consistent, they may be read/write consistent some of the time, but read/write consistency is not guaranteed all of the time. For example, the demonstration which concerns processes on different nodes using NFS without client caching is read/write consistent even though Process B uses client caching. This is because Process B reads the entire file at once. Processes on different clients using client caching or processes using *stdio* may be read/write consistent some of the time, but are not all of the time. The only way to guarantee read/write consistency for all cases of processes simultaneously accessing files and for all implementations is to use record locking and forgo the use of *stdio*.

3.2 Times Returned by *stat()*

The functions *stat()* and *fstat()* return the following times associated with a file:

1. *st_atime* - the time when data in the file was last accessed (e.g., by *read()*)
2. *st_mtime* - the time when data in the file was last modified (e.g., by *write()*)
3. *st_ctime* - the time when the status of the file was last changed (e.g., by *chmod()*)

Table 3.1: Read/write consistency summary

	Read/write consistent?
Process A and Process B on Same Node:	
Unix IO	Yes
Stdio	No
NFS	Yes
Process A and Process B on Different Nodes:	
NFS with no record locking	No
NFS with record locking	Yes

Unix implementations vary somewhat as to which functions update these times. In most implementations, the same basic set of functions cause a time associated with a file to be updated. For example, a *read()* of a file updates *st_atime* and a *write()* to a file updates both *st_mtime* and *st_ctime*. For a write, the file status, i.e., the file size, may change. However, some implementations have functions which update these times that other implementations do not have. For example, System V has a *lockf()* function and IEEE 1003.1-1990 does not.

Not only do implementations vary as far as which functions update the times returned by *stat()*, but implementations also vary in the procedure used to do the update. In some implementations, the times are updated when the function is performed, e.g., when a *read()* returns, the *st_atime* of the file has been set to the current time. In other implementations (including IEEE 1003.1-1990), the times returned by *stat()* are only updated periodically as a performance consideration. For example, when a *read()* is performed, the *st_atime* of the file is "marked for update." At some later time, e.g., when a *stat()* on the file is done, the "mark for update," left as a result of the most recent *read()*, causes the *st_atime* of the file to be set to the current time. For implementations which update *stat()* times in this manner, the values of the times are accurate only within the time period in which the "marks for update" are converted into time value changes. For implementations which update *stat()* times when a function is performed, the *stat()* times are as accurate as possible.

Most applications do not require accurate *stat()* times. One of the most common uses of *stat()* times is *make*. The *make* application is successfully used with almost all implementations.

The issue of *stat()* times becomes somewhat more difficult in a transparent file access environment. A file in a remote file system is physically located on a server. The server maintains its own clock which may or may not be synchronized with the client's clock.

If a client is caching to improve performance, then it is difficult to make accurate times available to processes on other clients. Reads and writes to remote files may actually only involve accessing the local caches. The server must be made aware of these accesses. There are solutions to these problems. The client could send a packet to the server when an access occurs or the server could poll its clients for updates whenever a *stat()* is done on a file. However, any solution which tries to maintain *stat()* times as accurately as can be done on a local file system incurs a performance penalty.

A more serious problem is that a server may not be able to maintain some or all of the *stat()* times because the native file system of the server does not have the equivalent concepts of *stat()* times. It is not uncommon for a file system to have only a file creation time.

3.3 Reporting of *write()* Error Conditions

When a *write()* is performed on a local file, any error condition associated with that *write()* is reported when the *write()* returns. However, it may not be possible for implementations of remote file systems, which cache data on the client and/or the server for performance considerations, to report all error conditions from *write()* when the *write()* returns. For example, the ENOSPC error condition from *write()* indicates that the file system is full. With a local file, this can be discovered easily when each *write()* takes place.

However, when the *write()* is to a remote file and the client implementation caches data for that file, it is not usually known until sometime later that the file system on the server, to which the file belongs, has filled. This is because the data from a *write()* to a remote file is actually written to the client cache. At some later time, the cache is emptied by sending data from several writes as a single packet to the server to be written to the file on the server's mass storage. At that time, if space on the server's file system fills, the server sends an indication to the client who then notifies the application which invoked the *write()*. Thus, an ENOSPC error condition may be reported to the application as an error condition returned to a *write()* other than the *write()* which actually caused the ENOSPC.

It may also happen that the ENOSPC error condition is reported on the *close()* of the file. This is potentially a more serious consequence for an application since most implementations do not now return an ENOSPC to a *close()*. As a result, most existing applications are not expecting an ENOSPC from a *close()*.

3.4 Record Locking

Record locking is a capability which is required for many applications, notably, for database applications. Many file system implementations support either advisory record locking and/or mandatory record locking. With advisory record locking, an application must check

to see if a record is locked before performing an operation on the record. Using advisory record locking, if an application accesses a locked record, the operation is permitted with no indication from the implementation that a lock was violated. This is in contrast to mandatory record locking where locks are enforced by the implementation. Using mandatory record locking, an application which attempts to violate a lock is refused access an error condition. In IEEE 1003.1-1990, record locking is advisory.

A record locking capability on either a local or a remote file system requires a file system implementation more complex than one that just provides input/output to files. For this reason and for performance considerations, many file system implementations do not provide record locking. For a remote file system, a record locking capability requires support from the client implementation, the server implementation, and the protocol. One of the goals of transparent file access is for a client to be able to have access to as many remote file systems as possible even if the access provided does not support all of the capabilities of the client file system. This implies that even if the client implementation supports record locking, that client may be accessing a server which cannot support record locking and/or may be accessing a server using a protocol which does not support record locking. In the case of a client using IEEE 1003.1-1990, not only must the server and the protocol support record locking, but the server and the protocol must support advisory record locking.

Chapter 4

Specific Environment Issues

Specific existing or emerging file systems may not support all of the file system semantics of IEEE P1003.1-1990. Problem areas include different file system structure (see sec. 4.1), different file attributes (see sec. 4.2), FIFOs in a network environment (see sec. 4.3), append mode *write()*s (see sec. 4.4), and the last-close semantic (see sec. 4.5).

4.1 Different File System Structures

On a single system, the file system usually has the same structural characteristics throughout. For example, on a file system with a hierarchical directory structure, directories are permitted anywhere the access permissions allow. On a file system that is *flat*, i.e., does not have directories, directories are permitted nowhere. On a single system, the same structure semantics usually apply to all accessible files regardless of location.

However, there are exceptions in the single system case. For example, some systems are capable of mounting different versions of their file systems. A flat file system may have been used in a system's early development. As the system matured, a hierarchical file system became the norm. In order to maintain backward compatibility, the system is able to simultaneously access both the early and the later versions of the file system. In such an environment on a single system, the structure semantics would be different depending on whether the part of the file system being accessed was the early version, i.e., the flat version, or the later version, i.e., the hierarchical version.

Systems which access file systems with different structure semantics become much more common in a network environment. Client file systems may have structure semantics different from server file systems.

Different structure semantics often implies a different syntax for naming files. For example, a flat file system likely has no way of recognizing a *pathname* as specified in P1003.1-1990 since a flat file system has no concept of a directory. The cover illustration shows three different file naming syntaxes, i.e., DOS and Unix on the clients, and VMS on the server. Some mapping between the client's file naming syntax and the server's file naming syntax may be required.

Different structure semantics does not prevent the client from being able to have some level of transparent access. Among the objects in a file system that make up its structure are directories, file types, and links.

4.1.1 Directories

Some file systems permit a hierarchical directory structure of virtually unlimited depth. Some file systems are flat and permit no directory structure at all. Some file systems permit directories but only up to a maximum depth. Other file systems permit directories but the directory structure is fixed or *read-only*, i.e., the creation or deletion of directories is not permitted. This is to be distinguished from a read-only file system in which neither files nor directories may be created or removed, and in which file data may neither be added nor modified. In a file system where the directory structure is read-only, files may be created and removed from directories but the directory structure remains static. Modification of the directory structure is an operation beyond the semantics of the file system. Some FTAM File Stores have such static directory structures.

In a network environment, a client may have several remote file systems attached from different servers and these file systems may vary in their directory manipulation capabilities. An application should be able to know the structure semantics of all accessible directories.

4.1.2 File Types

Some file systems support numerous file types and some file systems only support a few file types. For example, VMS supports variable length record sequential files, fixed length record sequential files, fixed length record random access files, index sequential files, stream files, and others. On the other hand, IEEE 1003.1-1990 supports only regular files (i.e., byte addressable random access files), character special files, block special files, FIFO special files, and directories. In a network environment, all file types on a client may not be supported on a server and likewise, all file types on a server may not be supported on a client. There is a serious problem if the client does not support any of the file types of the server. Transparent file access is reasonable as long as at least one file type on the client can be supported on the server. Otherwise, conversion between file types is required either by the client, by the server, or by the network.

4.1.3 Links

Links in a file system are additional directory entries or pointers to a file (i.e., those beyond the minimum required to establish the file's existence in the file system). Usually when the file is created, an entry is made in a directory. If the file system is flat, then the *directory* in this case is some *list of all files*. If another entry for the same physical file is created in the same or different directory, this additional entry is called a link. Consequently, the file is known by more than one name and may be accessed by more than one path through the directory tree. There are two types of links, hard links and symbolic links.

Usually, hard links are simply just multiple directory entries which point to the same physical file. The physical file usually consists of a *file header* and the data itself. The *file header* contains information about the file, such as, file access permissions and pointers to where the data is located. With hard links, it may not be possible to distinguish the directory entry created when the file was created from any other directory entry for the physical file. All directory entries pointing to the same physical file are *equal*.

On the other hand, a symbolic link is only a *virtual* link to a file. The physical file has a directory entry and may have hard links to it as well. A symbolic link to a file usually has a directory entry and a file header of its own. However, the file header for a symbolic link contains the name of a file to which the name of the symbolic link actually refers. Thus, given the name of a file which is a symbolic link, the system accesses the file pointed to by the symbolic link by obtaining the symbolic link name from the file header of the symbolic link. The symbolic link name is used to access the physical file. Symbolic links were developed in order to permit *links* to be made across different mounted file systems where the use of a hard link may not be feasible.

Some file systems support neither hard links nor symbolic links. Other file systems support links but the number of links to an individual file may be limited. In a network environment where several different file systems are mounted, an application may need to determine whether a hard or symbolic link can be made to a particular file, and how many links can point to a file.

4.2 Different File Attributes

File attributes are information associated with a file other than the actual data. File attributes may include such things as the owner of the file, the group ownership of the file, an access control list (i.e., a list of users who have access to the file), the file size, and a file serial number. Section 4.1 describes how a system in a network environment could have file systems mounted with different file system structures. Similarly, such a system could have file systems mounted where the files have different file attributes, or the same file attributes but with different semantics associated with the attributes. This should not prevent the files from being accessed transparently.

For example, in Unix and in VMS, there are access permission bits associated with owner access, group access, and access by others (called *world* access in VMS). However, the algorithm for permitting access based on these permission bits differs between Unix and VMS. In Unix, if the user trying to access a file is the owner, then the system checks only the owner permission bits. If the owner permission bits do not allow access to the file, then access is denied even if the owner is a member of the group and the group access is allowed, or access by others is allowed. In VMS, if the user trying to access a file is the owner and the owner permission bits deny access, then the system will check to see if the user can be granted access on the basis of group or world permissions.

4.2.1 File Access Control

Many operating systems, including Unix, are multiuser. As such, it is necessary to have mechanisms which protect a user's file from unwanted access. An integral part of an access control mechanism in a multiuser operating system is the concept of user ownership of a file and, very often, the concept of group ownership. However, many operating systems, such as those for personal computers, are single-user operating systems. In a single-user environment, there is no need for file access mechanisms which protect files from unwanted access by others because, conceptually, there is only one user. Nonetheless, some single-user operating systems permit a file to be marked as *read-only* to protect the file from inadvertent modification or removal.

Both single-user and multiuser operating systems have been used as servers providing transparent file access. Most multiuser operating systems are already well suited for use as a file server since access protection is already an integral part of its design. For a single-user operating system, it is necessary that some sort of access control mechanism be implemented when such an operating system is used to provide file service. This is usually accomplished by creating a separate partition for each user's files. Thus, user identification is not associated with an individual file but with a partition on the server's disk. File servers which have single-user operating systems may also have partitions that are accessible by anyone.

Network environments include both servers based on multiuser operating systems and servers based on single-user operating systems. Application programs on clients may have transparent access to servers which are based on single-user operating systems which may neither provide owner information nor group ownership information for a file, but are capable of imposing some level of file access control. That file access control may be no more than a read/write permission that applies to any user. A very large group of applications are able to function in such an environment.

4.2.2 Execute/Search Permissions

Unix and other file systems have the concept of an *execute/search* permission. If a file has execute permission, then the file is a program which may be loaded and run. If an attempt is made to run a file which does not have execute permission, then an error condition results. Execute/search permissions may also be used with directories. If a directory has search permission, then the directory can become the default directory or may be part of a complete path reference to a file. This is to be distinguished from a directory with read permission which means that the contents of the directory may be read.

Not all file systems have the concept of an execute/search permission. This is not so much of a problem if the server file system has execute/search permissions and the client does not. However, if the client has execute/search permissions and the server does not, then it will be difficult on the client to maintain the semantics associated with execute/search permissions. One solution is to not permit executable files to be run from a server whose file system cannot support execute/search permissions. Another solution is for the client to interpret the read permission as the execute/search permission.

4.2.3 Information Returned by *stat()*

The function *stat()* returns read/write and execute/search permissions for the file owner class, file group class, and file other class. Moreover, it returns the file's user ID and group ID plus several other things. As is noted in sections 4.2.1 and 4.2.2, a file system may not support user ID, group ID, and/or execute/search permissions. Consequently, *stat()* is unable to return meaningful information in those fields of the *stat()* structure.

In addition, it is possible that a file system may support the concept of user and group ownership of a file but the client's user/group identification information may differ from a server's user/group identification information. In such a case, in order to accomplish file access control between client and server, there must be a mapping of the client's user/group identification information to a server's user/group identification information.

For example, suppose Client A, Client B, and Server S all support IEEE 1003.1-1990. Client A, Client B, and Server S are each separately administered. The individual KO has an account on both clients and the server. On Client A, KO has a user ID of 10; on Client B, KO has a user ID of 15; on Server S, KO has a user ID of 20. The user KO is known by a different user ID on each of Client A, Client B, and Server S. Suppose Client A and Client B are using files on Server S. Server S, who knows KO as user ID 20, must be able to identify KO as user ID 10 when file access is attempted from Client A and as user ID 15 when access is attempted from Client B. Suppose Server S uses a simple mapping table which associates: (user ID 10, user ID 20) and (user ID 15, user ID 20). Now, Server S gets an access request for user ID 10, applies the mapping to get user ID 20, and then applies the file access control procedure based on a user ID of 20. Using the mapping, file access control works as it should for KO's files.

However, suppose an application on Client A performs a *stat()* on a file which has user ID 20 and is located on Server S. The server is unable to return meaningful information in the user ID part of the *stat()* structure. Server S cannot identify the file as owned by user ID 10 because the mapping is not one-to-one. The inverse mapping of user ID 20 gives both user ID 10 and user ID 15. Returning user ID 20, which the server knows as the owner of the file, wrongly identifies the file to the application on Client A. User ID 20 is not KO on Client A.

It is important to distinguish the functioning of the file access control mechanism of a file system and the ability of a file system to return meaningful information to a *stat()*. Note that, in the example, the file access control mechanism functioned as specified in IEEE 1003.1-1990 but the server was unable to return meaningful information about the file's owner. Most applications are capable of performing their primary function as long as they are able to access files. It is usually not necessary for an application to ascertain how the file access control mechanism functions. However, an application should be able to determine the meaningfulness of the information returned by *stat()*.

4.2.4 Attribute Manipulation

Most file systems permit the modification of file attributes after a file is created. However, some file systems do not. In some file systems, some attributes of directories cannot be changed. In some file systems, some attributes of files cannot be changed. Some file systems have times associated with file access which may not be changed without the access being made. FTAM file stores are examples of file systems which may not permit some attributes to be changed. Consequently, the IEEE 1003.1-1990 functions *chmod()*, *chown()*, and *utime()* may not be able to manipulate the supported file attributes (see sec. 4.2.1 and sec. 4.2.2) by a file system.

4.3 FIFOs in a Network Environment

A FIFO is a special file whose semantics differ significantly from those of a regular file. FIFO means *first-in-first-out*. The semantics of a FIFO can be summarized in terms of the operations on a regular file as follows:

1. Bytes written to a FIFO are always written as though they were written in append mode.
2. Bytes read from a FIFO are always read from the beginning of the file and then removed.
3. When all processes which have the FIFO open close the FIFO, any bytes remaining in the FIFO are removed.
4. Writing to a FIFO which no process has open for reading results in an error condition (see sec. 2.1).

Some file systems do not have the concept of a FIFO. In a network environment, a FIFO could be located on a server. Consequently, processes using such a FIFO could be located on different clients. Some file systems do no support the capability of processes on different clients sharing the use of a FIFO located on a server, i.e., the FIFO is interpreted as a local object. For such a file system, only processes on the same client may share the use of the remote FIFO even though it is located on a server. An application should be able to determine whether FIFOs are supported and whether a FIFO is interpreted as a local or as a remote object.

Demonstrations were developed using an NFS implementation to illustrate the behavior of a FIFO when it is located on a server and it is accessed by processes on the same client and processes on different clients. Figure 4.1 illustrates how the demonstration proceeds. Process A reads from the FIFO and Process B writes to the FIFO. Commands are displayed in *italics* and the output of those commands is displayed in **bold**. The FIFO *testfifo* is created on a remote file system mounted at the directory *mnt*.

Table 4.1 summarizes the results of using a FIFO on two different client/server configurations as follows:

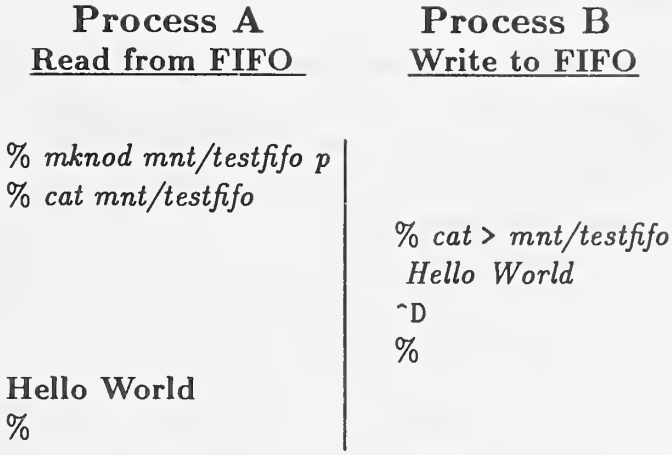


Figure 4.1: FIFO demonstration for processes on the same client.

Table 4.1: Summary of last-close semantic behavior using a FIFO located on an NFS server

Processes A&B run on	Does the FIFO work?
same client	Yes
different client	No

- Both processes using the FIFO are on the same client.

  Process A and Process B are both on Client A. Process A creates a FIFO on the server and issues a command to read from the FIFO. Process B writes to the FIFO. After Process B issues a ^D, both processes terminate successfully.

- The processes using the FIFO are on different clients.

  Process A is on Client A and Process B is on Client B. Process A creates a FIFO on the server and issues a command to read from the FIFO. Process B writes to the FIFO. Process A is unable to read from the FIFO. Neither process is able to terminate successfully.

4.4 Append Mode *write()*

Section 2.2 discussed how, in a network environment, it is more difficult to guarantee that when several processes simultaneously write to the same file, the block of data from one *write()* is not interleaved or overwritten by data from another *write()* before the first *write()* completes. The issue of non-interleaved writes of section 2.2 includes those cases where one or more of the writes are performed in append mode.

While the issue of non-interleaved writes is one which has implications for all file systems, the issue of non-interleaved writes in append mode by itself has implications for some file systems. Some remote file systems may not be able to guarantee multiple simultaneous append mode writes to the same file for reasons beyond those discussed in section 2.2. An application should be able to know if it is guaranteed non-interleaved append mode writes.

For example, in NFS, an append mode *write()* to a file is usually accomplished in two operations. First, the equivalent of a *stat()* is done on the file to ascertain the file size. Then, a write to the location indicated by the file size is performed. Since these two operations are not usually performed atomically, data from one append mode *write()* could overwrite data from another.

As an illustration of this, consider the following sequence of events. Process A on Client A writes 10 bytes to File X on the server in append mode. To accomplish this append mode write for Process A, Client A effectively does a *stat()* to the server to obtain the length of File X. In the meanwhile, Process B on Client B writes 5 bytes to File X on the server in append mode. Client B also effectively does a *stat()* to the server to obtain the length of File X. The server returns to both Client A and Client B the length of File X which is equal to 15. Client A now sends a request to the server to write 10 bytes at the sixteenth position of File X, thus appending its 10 bytes to the end of File X. The length of File X is now 25 bytes. Now, Client B sends its request to the server to write 5 bytes at the sixteenth position of File X. The 5 bytes written in append mode by Process B has overwritten the first 5 bytes of Process A's append mode write. Were Process A and Process B on the same system and File X a local file, File X would be of length 30 bytes and consist of its original 15 bytes, the 10 bytes written by Process A, and the 5 bytes written by Process B.

4.5 Last-close Semantic

The last-close semantic is the semantic which requires that an open file remain available to any process which has the file open regardless of any changes in file or process characteristics which may take place after the file is opened. It is called the *last-close* semantic because the best known consequence of the last-close semantic is that when a file is deleted, the file is not removed until the file is closed by the *last* process which has it open.

Table 4.2 lists IEEE 1003.1-1990 functions which have implications for the last-close semantic. Once a file has been successfully opened, the following conditions hold under IEEE 1003.1-1990 and traditional Unix for regular files. The function names listed below are IEEE 1003.1-1990 function names.

Table 4.2: IEEE 1003.1-1990 functions with last-close semantic implications

Change File		Change Process
Change Permission Bits and Ownership:	**Delete file:**	**Change Process Identity:**
`chmod()`	`unlink()`	exec file with set UID
`chown()`	`rename()`	exec file with set GID
		setuid()
		setgid()

1. Changing the file permission bits or ownership of a file does not cause any process with the file open to lose access.

   Once a process has a file open, access to the file is not denied. For example, if Process A has a file open and Process B uses *chmod* to change the file permission bits of the open file to 0, Process A does not lose access to the file. Despite the fact that a process can not open a file with mode 0, Process A does not lose access to the open file because file access modes are only checked when a file is opened. Unlike Unix, under IEEE 1003.1-1990, whether or not Process A would lose access to the open file is implementation-defined.

   The same example applies to *chown()*. If Process A has a file open and Process B uses *chown()* to change the ownership or group of the file, Process A retains access to the file.

2. Deleting the file by an *unlink()* or *rename()* does not cause any process with the file open to lose access.

   The function *unlink()* removes a link to a file and decrements the link count of the file. The file is no longer accessible when the file's link count is zero and no process has the file open. When an *unlink()* is performed and the link count becomes zero, the file is removed from the directory entry but the file contents are not removed until the last process which has the file open closes the file. Processes which have the file open, do not lose access to the unlinked file.

   A *rename()* can cause a file to be unlinked. A *rename()* which generates an implied *unlink()* of the file removes the directory entry for that file, but the removal of the file

contents is deferred until all references to the file have been closed and the link count is zero. If any processes had the unlinked file open, access to the file is not lost until the file is closed.

3. Changing the user ID and group ID of any process that has a file open does not cause that process to lose access to the file.

   A process can use the *exec* family of functions to overlay the current process image with a new process image. If Process A, which has a file open, calls *execl()* to execute *fileB*, the file descriptors open in Process A remain open. If *fileB* has its set-user-ID and/or set-group-ID mode bits set, then the effective user ID and/or the effective group ID of the process is changed. However, Process A does not lose access to open files inherited as a result of the *execl()*.

   If a process has appropriate privileges, the functions *setuid()* and *setgid()* can be used to set the user and group IDs for the process. Regardless of what the new user and group IDs are set to, access to an open file is not denied. File access modes are only checked when a file is opened.

To summarize, on a single system, access to a file by any process which has that file open is maintained regardless of any operation which changes the access conditions of the file (i.e. the owner, group, and access permissions), removes the file, or changes the effective user ID, effective group ID, or supplementary group IDs of the process. An exception to this is *chmod* under IEEE 1003.1-1990. Under IEEE 1003.1-1990, whether or not the last-close semantic is guaranteed for *chmod* is implementation defined.

The last-close semantic for directories is somewhat different than the last-close semantic for regular files. Directory streams are the file descriptor counterpart for directories in IEEE 1003.1-1990. IEEE 1003.1-1990 does not require that directories use file descriptors to implement directory streams. Note that if an implementation does not use file descriptors for directory streams, such file descriptors may not be used in functions, other than *readdir()*, *rewinddir()* , and *closedir()*, which have file descriptors as arguments (see IEEE 1003.1-1990 B5.1.2). As a result, directories may not have open file descriptions.

Moreover, the use of directory streams inherited from parent to child through *exec()* is undefined. IEEE 1003.1-1990 says that a directory stream inherited through a *fork()* may continue to be processed by either the parent or child but not by both. Thus, in a family tree of processes created by *fork()*, it is implementation-specific whether a directory steam in the root of the family tree is available to the root or to the processes which are the leaves of the family tree. If an inherited directory stream is usable by both parent and child, the result is undefined. Thus, if the last-close semantic is guaranteed for a directory, all open directory streams at the time a directory is removed are still available, but this availability among related processes may be considerably different from the application of the last-close semantic to regular files.

Consider the following examples:

1. Suppose the implementation is such that after a *fork()*, a directory stream continues to be available to the parent but not to the child. A process opens a directory D and creates a child using *fork()*. The directory stream for D is available to the parent but not the child. The parent removes D and the directory stream for D remains available to the parent. The availability of the directory stream for D was lost to the child as a result of the *fork()* and obviously, remains unavailable after D is removed by the parent.

2. Suppose that the implementation is such that after a *fork()*, a directory stream continues to be available to the child but not the parent. A process opens a directory D and creates a child by means of *fork()*. The directory stream for D is still available to the child but not to the parent. If the child removes D, then the directory stream for D remains unavailable to the parent even though the parent initially created the directory stream for D. The parent lost the availability of the directory stream for D after the *fork()*.

In these two examples, had the directory been a regular file, the file descriptors for D would have remained available to all processes. It is implementation-specific as to whether the system behaves like the first example or the second example. Thus, if the last-close semantic is not guaranteed, the only thing that is certain is that if the process that removes a directory has its directory stream available, it retains that availability. The availability of that directory stream to any other related process is implementation-specific.

The meaning of the last-close semantic for a FIFO special file is the same as for a regular file. However, there is an additional aspect of the last close semantic for a FIFO. The last-close semantic for a FIFO also implies that when all file descriptors associated with a FIFO have been closed, any remaining data in the FIFO is discarded. If the last-close semantic fails for a FIFO, data may remain in the FIFO after all file descriptors referring to the FIFO have been closed.

There are several common uses of the last-close semantic. For example, it is common for an application to create a temporary file and then immediately unlink the file. The temporary file remains available to the application but if the application is aborted, the temporary file is closed and removed without any further action by the application.

The last-close semantic is also sometimes used for process synchronization and message passing. The use of the last-close semantic for such applications was more common in earlier versions of Unix where the only facilities available for interprocess communication and synchronization were pipes and signals. Pipes and signals require that the processes who wish to communicate know each others process IDs and/or share open file descriptions. The following example illustrates how the last-close semantic can be used by two processes, who know nothing about each other, to pass messages of arbitrary length. The two processes only know a file name, *fileX*. This example assumes that *fileX* exists and had a mode of 666.

This message passing example does not queue messages. Messages are written and read one at a time. Process A writes a message of arbitrary size to a file, *fileX*, and Process B

reads a message in its entirety from *fileX*.

Process A and Process B proceed as follows (see programs *write_msg.c* and *read_msg.c* in Appendix B). When Process A wants to send a message to Process B, Process A checks to see if the length of *fileX* is zero. If the length is not zero, Process A waits for Process B to read the message and truncate *fileX* to length zero. If the length of *fileX* is zero, then Process A opens *file X* and uses *chmod()* to change the mode of *fileX* to zero. The chmod 0 prevents Process B from reading *fileX* before Process A has finished writing the message. Because of the last-close semantic, Process A does not lose access to *fileX*. When Process A has finished writing the message, it uses *chmod()* to restore the file permission bits of *fileX*.

When Process B wants to read a message from Process A, Process B checks to see if the length of *fileX* is 0. If the length is 0, Process B waits until there is a message to read. If the length of *fileX* is not zero, Process B proceeds to try to open *fileX*. If Process A has not finished writing the message, *fileX* will have mode 0 and Process B will have to wait for Process A to restore *fileX's* file permission bits before *fileX* can be opened. After Process B has read the message, it truncates *fileX* to indicate to Process A that it has read the message.

Demonstrations were developed to illustrate the behavior of the last-close semantic using NFS. These demonstrations are presented in the following sections. Section 4.5.1 demonstrates the behavior of the last-close semantic when an open file's ownership, group, and permissions bits are changed. Section 4.5.2 demonstrates the results of removing an open file. Section 4.5.3 shows the results of changing the identity of a process after a file has been opened. For the demonstrations, commands are displayed in *italics* and the output of those commands is displayed in **bold**.

4.5.1 Changing File Attributes

The program *read_file* (see Appendix B) opens a file, reads 2000 bytes, and then pauses until a character of input is received. When the program pauses, the file remains open and another process can modify file characteristics. The program *read_file* continues to read the file until end of file or a read error occurs. An error message is displayed if a read error occurs. For the following demonstrations, *testfile* is the file being read, Process A is the program *read_file*, and Process B is a process which changes the attributes of *testfile* when Process A pauses.

Note that before opening the file, the program *read_file* updates the access and modification times of the file. The purpose of this is to invalidate any caching of the file's data by the NFS client implementation. Such caching may yield different results for the demonstrations which illustrate the failure of the last-close semantic. The NFS client implementation is capable of caching entire files even when they are quite large. Updating the access and modification times of the file causes the client to go to the server to access the file rather than access the file in a local cache.

For all demonstrations, Process A is on Client A, Process B is either on Client A or Client B, and *testfile*, which is 38400 bytes, is remote to both Client A and Client B. The prompt for each process in the figures, e.g., "ClientA%", indicates on which client the process is running. On both Client A and Client B, the remote file system on which *testfile* resides, is

Process A *read_file*	Process B *chmod*
clientA% *ls -l mnt/testfile* **-rw-rw-rw- 1 procA_B 38400 mnt/testfile** clientA% *read_file mnt/testfile* **(read 2000 bytes from testfile)**	
	clientA% *chmod 0 mnt/testfile* clientA%
<return> **(read testfile until EOF)** **read 38400 bytes** clientA%	

Figure 4.2: Demonstration of the correct behavior of the last-close semantic using *chmod* with NFS.

mounted on the directory *mnt*. For the command "*ls -l*", the time of last modification for the file is omitted from the output displayed in the demonstrations. In some cases Process A is the owner of *testfile* (i.e., the effective user ID of Process A matches the user ID of *testfile*), and in other cases Process A is not the owner of *testfile*. If both Process A and Process B are owners of *testfile*, or if Process A is the owner and Process B has root privileges, the user ID of *testfile* is denoted as procA_B in the demonstrations. The user ID of *testfile* is procB if Process A is not the owner of *testfile* and Process B is either the owner of *testfile* or has root privileges to modify *testfile*.

Table 4.3 summarizes the effect on Process A's ability to maintain access to *testfile* after Process B uses *chmod* to change the mode of *testfile* to 0 as described below:

- Process A is the owner of *testfile* (see fig. 4.2 which illustrates the results where Process A and Process B are on the same client).

  Process A opens *testfile*, reads 2000 bytes, and pauses. Process B changes the mode of *testfile* to 0. Process A successfully continues to read *testfile*. Even though Process B has changed the mode to 0, Process A retains access to *testfile* regardless of whether Process B is on the same client or a different client.

- Process A is not the owner of *testfile* (see fig. 4.3 which illustrates the results where Process A and Process B are on different clients).

  Process A opens *testfile*, reads 2000 bytes, and pauses. Process B changes the mode of *testfile* to 0. Process A continues to read *testfile* until an error occurs. Process A does not retain access to *testfile* after Process B changes the mode to 0 regardless of whether

Process A *read_file*	**Process B** *chmod*
clientA% *ls -l mnt/testfile* **-rw-rw-rw- 1 procB 38400 mnt/testfile** clientA% *read_file mnt/testfile* **(read 2000 bytes from testfile)**	
	clientB% *chmod 0 mnt/testfile* clientB%
<return> **(read testfile until an error occurs)** **13-Read: Permission denied** **read 16384 bytes** clientA%	

Figure 4.3: Demonstration of the failure of the last-close semantic using *chmod* with NFS.

Process B is on the same client or a different client. If either client is using caching, the effects of changing the mode of *testfile* may not appear instantly on Client A. Data in Client A's cache is still readable. This accounts for Process A's ability to continue to read data for some time after Process B has changed the access permissions. When the data in the cache is exhausted, the read error occurs. A file of 38400 bytes is used to demonstrate that the last-close semantic is not guaranteed. If *testfile* is small enough so that the remaining data can fit into cache, the last-close semantic may be met, but the last-close semantic is not guaranteed for all cases where Process A is not the owner of *testfile*.

Table 4.3 shows that using *chmod* with NFS, the last-close semantic is guaranteed when the process reading a remote file is the owner of that file. It does not matter whether the process reading the file and the process performing the *chmod* are on the same system. If the process reading the file is not the file owner, the last-close semantic is not guaranteed.

Table 4.4 summarizes the effect on Process A's ability to maintain access to *testfile* after Process B uses *chown* to change the owner of *testfile* as described below:

- Process A is initially the owner of *testfile* (see fig. 4.4 which illustrates the results where Process A and Process B are on different clients).

  For this demonstration, Process A is always initially the owner of *testfile*. The remote file *testfile* has no group or other permission bits set. Process A opens *testfile*, reads 2000 bytes, and pauses. Process B, which has root privileges, uses *chown* to change the owner of *testfile* to procB. Note that the client where Process B is located must

Table 4.3: Summary of last-close semantic behavior using *chmod* with NFS

Is Process A File owner?	Processes A&B run on	Last-close semantic guaranteed?
Yes	same client	Yes
Yes	different client	Yes
No	different client	No
No	same client	No

have the remote file system where *testfile* resides mounted with root access. Process A continues to read *testfile* until an error occurs. Regardless of whether Process B is on the same client or a different client, Process A does not retain access to *testfile* after Process B changes the ownership of *testfile*. If either client is using caching, the effect of the change in ownership of *testfile* may not appear instantly on Client A. Data in Client A's cache is still readable. This accounts for Process A's ability to continue to read data for some time after Process B has changed the ownership of *testfile*. When the data in the cache is exhausted, the read error occurs.

Table 4.4: Summary of last-close semantic behavior using *chown* with NFS

Is Process A File owner?	Processes A&B run on	Last-close semantic guaranteed?
Yes	same client	No
Yes	different client	No

Table 4.4 shows that with NFS the last-close semantic is not guaranteed for *chown*. Unlike the *chmod* demonstration, the cases where Process A is not the owner of *testfile* do not apply. The *chown* demonstration relies on the fact that Process A initially has access to *testfile* because Process A is the file's owner. If *testfile* were opened based on group or other

Process A *read_file*	Process B *chown*
clientA% *ls -l mnt/testfile* **-rw------- 1 procA_B 38400 mnt/testfile** clientA% *read_file mnt/testfile* **(read 2000 bytes from testfile)**	
	clientB# *chown procB mnt/testfile* clientB#
<return> **(read testfile until an error occurs)** **read 16384 bytes** **13-Read: Permission denied** clientA%	

Figure 4.4: Demonstration of the failure of the last-close semantic using *chown* with NFS.

permissions, i.e., Process A was not the owner of *testfile*, Process A would still be able to access *testfile* on the basis of group or other access.

Table 4.5 summarizes the effect on Process A's ability to maintain access to *testfile* after Process B uses *chgrp* to change the group of *testfile* as described below:

- Process A is not the owner of *testfile*, but initially has group access (see fig. 4.5 which illustrates the results where Process A and Process B are on different clients).

  For this demonstration, Process A is never the owner of *testfile*. The file *testfile* has a group ID of groupA. Process A, which has a group ID of groupA, opens *testfile* reads 2000 bytes of *testfile*, and pauses. Process B, which has root privilege, uses *chgrp* to change the group of *testfile* to groupB, a group of which Process A is not a member. Note that the file system on which *testfile* is located must be mounted with root access. Process A continues to read *testfile* until an error occurs. Process A does not retain access to *testfile* after Process B changes the group of *testfile* regardless of the location of Process B. If either client is using caching, the effect of the change in the group of *testfile* may not appear instantly on Client A. Data in Client A's cache is still readable. This accounts for Process A's ability to continue to read data for some time after Process B uses *chgrp* to change the group of *testfile*. When the data in the cache is exhausted, the read error occurs.

Table 4.5 shows that using NFS the last-close semantic is not guaranteed for *chgrp*. Unlike the *chmod* demonstration, the cases where Process A is the owner of *testfile* do not apply because if Process A opened *testfile* based on Process A's ownership, Process A would

Process A *read_file*	Process B *chgrp*
clientA% *ls -lg mnt/testfile* **----rw---- 1 procB groupA 38400 mnt/testfile** clientA% *read_file mnt/testfile* **(read 2000 bytes from testfile)**	
	clientB# *chgrp groupB mnt/testfile* clientB#
<return> **(read from testfile until an error occurs)** **read 16384 bytes** **13-Read: Permission denied** clientA%	

Figure 4.5: Demonstration of the failure of the last-close semantic using *chgrp* with NFS.

maintain access to *testfile* regardless of the change in group. For the same reason, cases where *testfile* has other permission bits set do not apply to the *chgrp* demonstration. Note that in the case where Process A is the owner of *testfile* but the file access permissions do not allow owner access, Process A would not be able to open *testfile* even if Process A had group or other access.

The behavior of the demonstrations in this section is a result of the fact that NFS is a *stateless* protocol, i.e., an NFS server does not know that a client has a file open. Since the server can not know that a client has a file open, the NFS client is responsible for maintaining the last-close semantic for files accessed using NFS. Reads and writes of a remote file by a client in NFS are made by sending requests to an NFS server. Each request includes the user ID and group ID of the requesting process. Access to a file is checked against the file's user

Table 4.5: Summary of last-close semantic behavior using *chgrp* with NFS

Is Process A File owner?	Processes A&B run on	Last-close semantic guaranteed?
No	different client	No
No	same client	No

ID, group ID, and file access permissions by the NFS server when each request is received. Changes in a file's user ID, group ID, or file access permissions may change while a client has the file open. Read or write requests received after the change, which previously would have been accepted, are now rejected. A common method used by NFS servers to assist clients in maintaining the last-close semantic is to always permit access by a file's owner (i.e., the process whose effective user ID matches the user ID of a file). This accounts for the behavior of the demonstrations in this section which manipulate the file attributes of open files related to access rights.

4.5.2 Deleting a File

Demonstrations were developed to illustrate the behavior of the last-close semantic using NFS when a remote open file is deleted. The program *read_file*, as described at the beginning of section 4.5.1, is also used in this section. The demonstrations in this section proceed in the same manner as those in section 4.5.1. Process A is the program *read_file*, and Process B is either the command *rm* or *mv*. The commands *rm* and *mv* are used to delete *testfile*.

Table 4.6 summarizes the effect on Process A's ability to maintain access to *testfile* after Process B uses *rm* to remove *testfile* as described below:

- Process A and Process B are on the same client (see fig. 4.2 replacing the command "chmod 0 mnt/testfile" with "rm mnt/testfile").

  Process A and Process B are on Client A. Process A opens *testfile*, reads 2000 bytes, and pauses. Process B uses *rm* to remove *testfile*. Process A successfully continues to read *testfile* regardless of whether Process A is the owner of *testfile*.

- Process A and Process B are on different clients (see fig. 4.6).

  Process A is on Client A and Process B is on Client B. Process A opens *testfile*, reads 2000 bytes, and pauses. Process B uses *rm* to remove *testfile*. Process A continues to read *testfile* until an error occurs. Process A does not retain access to *testfile* after Process B removes *testfile* regardless of whether Process A is the file's owner. If caching is used, then access to *testfile* may not be denied immediately. Data in Client A's cache is still readable. This accounts for Process A's ability to continue to read data for some time after Process B removes *testfile*. When the data in the cache is exhausted, the read error occurs.

Table 4.6 shows that the last-close semantic is not guaranteed for all cases of *rm*. Compare table 4.6 to table 4.3, table 4.4, and table 4.5 of section 4.5.1. For the tables of section 4.5.1, whether the last-close semantic is maintained for an open file when its file access characteristics are changed is a function of whether the opening process maintains ownership of the open file. If a process maintains ownership of a file, then the last-close semantic is guaranteed for the process accessing the open file regardless of changes in file access permissions. On the other hand, table 4.6 shows that file access is maintained only if Process A and Process

Process A *read_file*	Process B *rm*
clientA% *ls -l mnt/testfile* **-rw-rw-rw- 1 procA_B 38400 mnt/testfile** clientA% *read_file mnt/testfile* **(read 2000 bytes from testfile)**	
	clientB% *rm mnt/testfile* clientB%
<return> **(read from testfile until an error occurs)** **read 16384 bytes** **70-Read: Stale NFS file handle** clientA%	

Figure 4.6: Demonstration of the behavior of the last-close semantic using *rm* with NFS.

B are on the same client. The ownership of *testfile* is immaterial. As pointed out at the end of section 4.5.1, the results of table 4.3, table 4.4, and table 4.5 are a consequence of the fact that an NFS server helps a client in maintaining the last-close semantic by always granting access to the owner of a file. In the case of *rm*, the server is unable to provide a client any assistance in maintaining the last-close semantic.

Table 4.6 shows that the last-close semantic is guaranteed when the remote open file is deleted on the same client which has the file open. When both processes are on the same client, the implementation on the client may take steps to give the appearance that the file is not removed immediately. For example, using NFS, when Process A and Process B are both on Client A, Client A knows that Process A has *testfile* open. When Process B initiates a command to remove *testfile*, Client A renames *testfile* on the server to a file called *.nfsxxx* where *xxx* is a number. Process A uses the *.nfsxxx* file as though it were *testfile*. When Process A closes *testfile* (now *.nfsxxx*), Client A removes the *.nfsxxx* file as long as no other process has it open. The *.nfsxxx* file may be seen by doing an "ls -a" on the directory where *testfile* is located. For an NFS implementation, when Process A and Process B are on the same client, last-close semantics are maintained.

Tables 4.7 and 4.8 summarize the effect on Process A's ability to maintain access to *testfile* after Process B uses *mv* to move the file *junk* to the file *testfile*, thus deleting the original *testfile*. The procedure is described as follows:

- The file *junk* is remote and is on the same remote file system as *testfile* (See fig. 4.6 replacing the command "rm mnt/testfile" with the command "mv mnt/junk mnt/testfile". Figure 4.3 illustrates the results of a demonstration where Process A and Process B

Table 4.6: Summary of last-close semantic behavior using *rm* with NFS

Is Process A file owner?	Processes A&B run on	Last-close semantic guaranteed?
Yes	same client	Yes
Yes	different client	No
No	different client	No
No	same client	Yes

are on different clients and Process A is the owner of *testfile*).

Process A opens *testfile*, reads 2000 bytes, and pauses. Process B uses *mv* to move *junk* to *testfile*. Process A continues to read *testfile* until an error occurs. If caching is used, data in Client A's cache is still readable. When the data in the cache is exhausted, the read error occurs. Process A does not retain access to *testfile* after Process B unlinks *testfile* regardless of whether Process A is the file's owner or whether Process A and Process B are on the same or different clients.

Table 4.7: Summary of the failure of the last-close semantic using *mv* with NFS

Location of *junk* in relation to Client A	Does Process A own *testfile* ?	Processes A&B run on	Last-close semantic guaranteed?
same remote filesystem as testfile	Yes	same client	No
same remote filesystem as testfile	Yes	different client	No
same remote filesystem as testfile	No	different client	No
same remote filesystem as testfile	No	same client	No

- The file *junk* is local (see fig. 4.2 replacing the command "chmod 0 mnt/testfile" with the command "mv junk mnt/testfile").

Process A and Process B are both on Client A. Process A opens *testfile*, reads 2000 bytes, and pauses. Process B uses *mv* to move the local file *junk* to the remote file *testfile*. Process A successfully continues to read *testfile* regardless of whether Process A is the owner of *testfile*.

- The file *junk* is remote but is on a different remote file system than *testfile* (see fig. 4.2 replacing the command "chmod 0 mnt/testfile" with the command "mv mnt2/junk mnt/testfile").

  Process A and Process B are both on Client A. Client A has the file system containing *junk* mounted under the directory mnt2. Process A opens *testfile*, reads 2000 bytes, and pauses. Process B uses *mv* to move the remote file *junk* to the remote file *testfile*. Process A successfully continues to read *testfile* regardless of whether Process A is the owner of *testfile*.

Table 4.8: Summary of last-close semantic behavior using *mv* with NFS

Location of *junk* in relation to Client A	**Does Process A own *testfile* ?**	**Processes A&B run on**	**Last-close semantic guaranteed?**
local	Yes	same client	Yes
local	No	same client	Yes
different remote file-system than testfile	Yes	same client	Yes
different remote file-system than testfile	No	same client	Yes

Table 4.7 summarizes the results of using *mv* to move *junk*, which in this case is on the same remote file system as *testfile*, to *testfile*, thus removing *testfile*. The last-close semantic is not guaranteed for all cases where *junk* is on the same remote file system as *testfile* regardless of whether Process A is the owner, or whether Process A and Process B are on the same or on a different client.

Using NFS with the files *junk* and *testfile* on the same remote file system, *mv* behaves similarly to *mv* on a local system. The file *testfile* is unlinked, a link is made from *junk* to *testfile*, and then *junk* is unlinked. On a local system, the last-close semantic is guaranteed and *testfile* is not removed until the link count is zero and no processes have the file open. Using NFS, the server has no way of knowing that the client has *testfile* open so the original *testfile* is removed immediately, thus causing the failure of the last-close semantic as summarized in table 4.7. It is possible for a client NFS implementation to maintain the last-close

semantic when Process A and Process B are on the same client. The technique used would be the same used in the case of *rm*, i.e., an *.nfsxxx* file could be created when *testfile* is effectively unlinked by the *mv*.

Table 4.8 summarizes the results of using *mv* to move *junk*, which is either local to Client A or on a different remote file system than *testfile*, to *testfile* when Process A and Process B are on the same client. Using NFS, when the source and destination files for a *mv* are on different file systems, *mv* behaves differently than if the source and destination files were both local. Because *junk* is being moved across different remote file systems, *testfile* is unlinked, *junk* is copied to *testfile*, and then *junk* is unlinked from the file system on which it was previously located. Note that *junk* is copied to *testfile* instead of a link being made from *junk* to *testfile*.

For the cases summarized by table 4.8, the NFS implementation on Client A takes steps to give the appearance that *testfile* is not removed immediately when Process B causes *testfile* to be unlinked. Client A knows that Process A has *testfile* open. Client A copies *testfile* on the server to a file called *.nfsxxx* before removing *testfile*. The file name suffix *xxx* is a number. Process A uses the *.nfsxxx* file as though it were *testfile*. The *.nfsxxx* files may be seen by doing an "ls -a" on the directory where *testfile* is located. When Process A closes *testfile* (now *.nfsxxx*), Client A removes the *.nfsxxx* file as long as no other process has it open.

Table 4.8 shows that the last-close semantic is guaranteed for cases where Process A and Process B are on the same client, and *junk* is either local or on a different remote file system than *testfile*. For those cases where Process A and Process B are on different clients, the last-close semantic is not guaranteed (see comments on table 4.7). The results are not included in table 4.8 in the interest of minimizing the complexity of the table.

In section 4.5.1, the demonstrations using NFS illustrated that when file attributes of open files are changed, the last-close semantic is maintained only in those cases where the process accessing the file was the owner of the file. The conclusion from table 4.6 of this section is that, using NFS, when an open file is deleted the last-close semantic is only maintained in those cases where the two processes are on the same client. Tables 4.7 and 4.8 show that, using NFS, when an open file is deleted as a result of a rename, the last-close semantic is only maintained in those cases where the source file is on a different file system from the destination file and the two process are on the same client. For both *rm* and *mv* using NFS, the last-close semantic fails in all cases where the two processes are on different clients.

4.5.3 Changing Process Identity

Demonstrations were developed using NFS to illustrate the behavior of the last-close semantic when a process's identity is changed. Programs, which are modifications of *read_file*, were written to determine the behavior of the last-close semantic when a process, Process A, changes its process identity either by using *exec* to execute a file with its set-user-ID bit or set-group-ID bit set, or by using *setuid()* or *setgid()*. Process A is either the program *read_n_exec*

Process A
read_n_exec

clientA% *read_n_exec mnt/testfile set_user_ID*
(read 2000 bytes from testfile)
(exec the file set_user_ID)
(read testfile until end of file)
read 38400 bytes
clientA%

Figure 4.7: Demonstration of last-close semantic behavior using *exec* to execute a file with set-user-ID bit or set-group-ID bit set.

or *read_n_changeID*. These programs and the program *set_user_ID* which is executed by *read_n_exec* are listed in Appendix B. The program *set_group_ID* is not include in Appendix B because it is identical to *set_user_ID* with the exception that *set_user_ID* has its set-user-ID bit set and *set_group_ID* has its set-group-ID bit set. The program *read_n_exec* accepts an argument for the name of a file to read and an argument for the name of the file to execute. The program *read_n_changeID* accepts three arguments: the name of a file to read, the type of ID to change, and a number which, depending on the previous argument, is either the user ID or group ID the process is to become. For all the demonstrations, Process A, which is located on Client A, reads the remote file *testfile*, which is 38400 bytes and is located on a remote file system mounted on the directory mnt.

Table 4.9 summarizes Process A's ability to maintain access to *testfile* after Process A changes its process identity according to the procedures described below:

- *exec* a file with its set-user-ID bit set (see fig. 4.7).

  Process A is the program *read_n_exec* with parameters *mnt/testfile* and *set_user_ID*. The user ID of Process A is 100. The file *testfile* only permits owner access and is owned by Process A (i.e., the owner ID of *testfile* is 100). Process A opens *testfile*, reads 2000 bytes, and uses *exec* to execute the file *set_user_ID*. The program *set_user_ID*, which has an owner ID of 199 and has its set-user-ID bit on, successfully continues to read *testfile*. Process A does not lose access to *testfile* even though its effective user ID changes to 199, a user ID that would not normally be able to open *testfile*. Using NFS, the last-close semantic is guaranteed for cases where a process executes a file with its set-user-ID bit set.

- *exec* a file with its set-group-ID bit on (see fig. 4.7 replacing the parameter set_user_ID with the parameter set_group_ID).

  Process A is the program *read_n_exec* with parameters *mnt/testfile* and *set_group_ID*. The file *testfile* is not owned by Process A and does not permit owner or other access,

Process A
read_n_changeID

clientA% *read_n_changeID mnt/testfile user 199*
(read 2000 bytes from testfile)
(call setuid())
(read testfile until end of file)
read 38400 bytes
clientA%

Figure 4.8: Demonstration of last-close semantic behavior using *setuid()* to change process identity.

but has a group ID of 200 and permits group access. Note that in the case where Process A is the owner of *testfile* but the file access permissions do not allow owner access, Process A would not be able to open *testfile* even if Process A had group or other access. Process A, which has a group ID of 200, opens *testfile*, reads 2000 bytes, and then uses *exec* to execute the file *set_group_ID*. The program *set_group_ID*, which has its set-group-ID bit on and a group ID of 199, successfully continues to read *testfile*. Process A does not lose access to *testfile* even though its effective group ID is changed to 199, a group that would not normally be able to access *testfile*. Using NFS, the last-close semantic is guaranteed for cases where a process executes a file with its set-group-ID bit set.

- *setuid()* (see fig. 4.8).

  Process A is the program *read_n_changeID* with parameters *mnt/testfile*, *user*, and *199*. The file *testfile* has an owner ID of 100 and only permits owner access. The program *read_n_changeID* must be owned by the superuser and have its set-user-ID bit on in order for Process A to be able to use *setuid()*. The file system containing *testfile* must be mounted with root access so that Process A, whose effective user ID is the superuser, can initially open *testfile*. Process A opens *testfile*, reads 2000 bytes, determines which ID to change, and then uses *setuid()* to change the real and effective user IDs of *testfile* to 199. Process A continues to read *testfile* even though its effective user ID is changed to 199, a user ID that would not normally be able to access *testfile*. Using NFS, the last-close semantic is guaranteed for cases where a process uses *setuid()* to change its real and effective user IDs.

- *setgid()* (see fig. 4.8 replacing the parameter *user* with the parameter *group*).

  Process A is the program *read_n_changeID* with parameters *mnt/testfile*, *group*, and *199*. The file *testfile* has a group ID of 200 and only permits group access. The program *read_n_changeID* must be owned by the superuser and have its set-user-ID bit on in

order for Process A to be able to use *setgid()*. The file system containing *testfile* must be mounted with root access so that Process A can initially open *testfile*. Process A opens *testfile*, reads 2000 bytes, and then uses *setgid()* to change the real and effective group IDs of *testfile* to 199. In addition to the change of the group ID of *testfile*, the effective user ID is set to the real user ID so that the effective user ID of Process A is no longer that of the super-user. Process A continues to read *testfile* even though its effective group ID is changed to a group ID that would not normally be able to access *testfile*. Using NFS, the last-close semantic is guaranteed for cases where a process uses *setgid()* to change its real and effective group IDs.

Table 4.9 shows that using NFS, the last-close semantic is guaranteed when a process changes its process identity by using *exec* to execute a file with its set-user-ID bit or set-group-ID bit set, or by using *setuid()* or *setgid()*. Because NFS is stateless, the NFS server on which *testfile* resides does not know whether Client A has *testfile* open. Therefore, the server must do permission checking each time Client A issues a read request to read from *testfile*. Client A adds authentication information including the effective user ID and group ID of Process A to each read request sent to the server. When the server receives each request to read *testfile*, the server checks Process A's effective user ID and group ID against *testfile's* user ID, group ID, and file access permissions to determine if access to *testfile* will be granted.

Table 4.9: Summary of last-close semantic behavior when changing process identity

Change Process Identity	**Last-close semantic guaranteed?**
exec file with set UID	Yes
exec file with set GID	Yes
setuid()	Yes
setgid()	Yes

Client A gets the effective user ID and group ID to include in the read request from its file descriptor table. The effective user ID and group ID contained in the file descriptor table are the effective user ID and group ID of the process when *testfile* is opened. The file descriptor table does not reflect changes in the process's identity. The program *show_IDs* is a modification of *read_n_changeID* which does not display data that is read from *testfile*, but

displays the effective user ID and group ID stored in the process table and stored in the file descriptor table entry for *testfile*. Source code for *show_IDs* is located in Appendix B.

Figure 4.9 shows the output from *show_IDs* given the parameters *mnt/testfile*, *user*, and *199*. The program *show_IDs* has its set-user-ID bit set and is owned by the super-user. Figure 4.9 also shows that for this demonstration *testfile* is 38400 bytes, only permits owner access, has an owner ID of 100, and a group ID of 200. When *show_IDs* changes it process identity by setting its user ID to 199, figure 4.9 shows that the change in process identity is not reflected in the file descriptor table entry for *testfile*. However, the change in process identity is reflected in the process table. If Client A got the effective user ID and group ID to include in the read request from the process table instead of the file descriptor table, the last-close semantic would not be maintained.

Process A

show_IDs

clientA% *ls -lg mnt/testfile*
-rw------- 1 100 200 38400 mnt/testfile
clientA% *show_IDs mnt/testfile user 199*
Before setuid() or setgid():

Information from the process table:
The effective uid is 0:
The effective gid is 200:

Information from the file descriptor table:
The effective uid of the user who opened mnt/testfile is 0:
The effective gid of the user who opened mnt/testfile is 200:

Changed the real and effective user IDs to 199

After setuid() or setgid():

Information from the process table:
The effective uid is 199:
The effective gid is 200:

Information from the file descriptor table:
The effective uid of the user who opened mnt/testfile is 0:
The effective gid of the user who opened mnt/testfile is 200:

read 38400 bytes
clientA%

Figure 4.9: Demonstration which shows the effective UIDs and GIDs of the program *show_IDs*.

Chapter 5

Miscellaneous Issues

Miscellaneous issues are those which do not clearly fit in one of the other categories. Miscellaneous issues include problems associated with supplementary groups (sec. 5.2), the set-user/group-ID capability (sec. 5.4), and devices in a Network Environment (sec. 5.5). Miscellaneous issues also include those problem areas which have implications for all other categories. These include new error conditions (sec. 5.1) and file location (sec. 5.3).

5.1 New Errors

In accessing files across a network, file access error conditions are more common. For example, the EIO error conditions would occur much more frequently for remote files than for local files. Consequently, an application which does not check for EIO or simply fails completely when an EIO occurs, may have to be modified so that it can continue.

The EIO error condition indicates an unrecoverable situation. Access to a file or device is lost forever. In traditional Unix and IEEE 1003.1-1990, the EAGAIN error condition indicates that access to a file or device may be only temporarily lost. The application may regain access by trying again using the same file descriptor.

In NFS, ESTALE is also used to indicate a recoverable access error condition. ESTALE in an NFS implementation means that an identification code, the NFS "file handle," has lost its validity. Consequently, the validity of any file descriptors associated with an invalid file handle have been lost. One way that this may occur is a server crash. When the server recovers, file handles which an application may have been using may no longer be valid. New file handles may be obtained by closing and opening the file.

In addition to requiring an error condition for indicating another kind of recoverable file access error, transparent file access also needs an error condition to indicate that a particular type of file access may not be supported. For example, a client whose local file system supports directories, i.e., a hierarchical file system, may be accessing a file system on a server which does not support directories, i.e., the server only supports a "flat" file system. A collection of files from such a server may be mounted in a directory on the client but the client cannot create a directory in the remote file system. In this case, an application should

expect an error condition, such as EOPNOTSUPP, indicating "operation not supported" when attempting to create a directory.

5.2 Number of Supplementary Groups

In the IEEE 1003.1-1990 Standard, associated with a process are its effective user ID, effective group ID, and a set of supplementary group IDs. Associated with a file is a group ID which indicates the group ownership of the file. The file access mechanism of the IEEE 1003.1-1990 Standard includes checking the effective group ID of the process, as well as, each of the supplementary group IDs of the process, against the group ownership of the file to see if the process has access to a file based on group permissions. If a process is not the owner of a file, then access to a file is granted to the process if its effective group ID or any of its supplementary group IDs match the group ownership of the file, and if the file group access permissions are set for the operation that the process wishes to perform. The *sysconf()* variable NGROUPS_MAX indicates to an application the maximum number of supplementary groups permitted by the implementation.

In a network environment, the value of NGROUPS_MAX on a client may differ from the value of NGROUPS_MAX on a server. In particular, NGROUPS_MAX for the client may be greater than NGROUPS_MAX for the server. The implementation should deal with this situation in such a manner that a process on the client is not denied access to a file that it should be able to access based on one of its supplementary groups. For example, an implementation could check access rights based on supplementary group membership by accessing the server several times giving the server each time the number of groups that the server can accept. If the client knows that the process is a member of the group which owns the file, then by always including that group in the list given the server, correct access verification is assured. In addition, an application should be able to obtain from the implementation the maximum number of supplementary groups permitted by a file.

5.3 File Location

In Unix, an application accesses a file by referencing a pathname in a single directory tree which contains all of the names of the files in the file system. In an environment where an application has access to several different file systems, which may be attached at any position of the directory tree, it should be able to ascertain whether a file is physically located on the local system or on a remote server. From an implementation point of view, the physical location of a file is usually the system on whose disk the inode for the file resides.

5.4 Set-User/Group-ID Capability

Each executable file in a Unix file system is associated with a set-user-ID bit and a set-group-ID bit. If the set-user-ID bit and/or the set-group-ID bit is set, then the effective user

ID and/or group ID of the process is set to the owner ID and/or group ID of the executable file upon execution.

In a network environment, if an executable file is part of a file system which supports execute/search permissions (see sec. 4.2.2), the administrator of a client may have disabled the set-user/group-ID capability as a security precaution. An application should be able to determine for a particular executable file whether the file's set-user/group-ID capability is functional.

5.5 Devices in a Network Environment

In a Unix file system, the input/output devices, such as, terminals, printers, tape, and disk drives, are referenced as files in the directory tree which contains the names of all the files in the file system. All devices are usually located in the */dev* directory. For example, a terminal device can be accessed by referencing a name usually of the form *ttyn* where *n* is some number. If an application wishes to read or write to a terminal device, it uses the name */dev/ttyn* as the pathname argument to Unix I/O functions.

Some file systems may not be able to support devices as part of a the file system's directory tree. For those file systems that can support devices, the use of a terminal device as a controlling terminal may not be supported. In a network environment, where an application may have access to several different file systems, the application should be able to obtain the following information associated with devices:

- Whether the file system supports devices.
- If devices are supported, whether the physical location of the device, i.e., the system to which the device is physically attached, is the local system or a server.
- If devices are supported, whether a particular terminal device can become a controlling terminal.
- If devices are supported, whether the location of the device file, i.e., where the inode resides, is on the local system or a server.

Chapter 6

Conclusion

This report has presented many issues and problems which arise when the IEEE 1003.1-1990 Standard is applied to environments other than the one for which it was developed, i.e., the single system accessing a local file system. The issues and problems discussed are those whose resolution formed the basis of the IEEE 1003.8 Transparent File Access (TFA) Standard which is under development. This specification provides:

- A standard way of characterizing and profiling file systems.
- Access to the widest possible kinds of file systems which can resemble the file system of IEEE 1003.1-1990.
- The means for an application program to simultaneously manipulate files whose access characteristics differ.

With examples and demonstrations using NFS, the most widely used and implemented remote file system, this report has illustrated some of the features and capabilities of Unix file systems, in particular, the file system specified in IEEE 1003.1-1990. Many of these features, e.g., the last-close semantic and non-interleaved writes, are not required for most applications. Most applications, e.g., word processors, consist of a single process simply reading and writing files without shared access with any other process. On the other hand, a database application usually consists of several processes on several systems simultaneously accessing the same database files. For such an application, the file system which it accesses must be robust enough to support the shared simultaneous access by many processes to many files.

The IEEE 1003.8 TFA Standard under development is a specification which is suitable for use by all applications. For the simple application, such as a word processor, the IEEE 1003.8 TFA Standard provides an access specification for rudimentary file systems, while, at the same time, provides an access specification for a robust file system capable of supporting complex applications, such as a database management system.

Appendix A

References and Related Reading

Balkivich, E., Lerman, S., Parmelee, R. P., "Computing in Higher Education: The Athena Experience," *Communications of the ACM*, November 1985.

Birrell, A. D., Nelson B. J., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, February 1984.

Champine, G. A., Geer, D. E., Ruh, W. N., "Project Athena as a Distributed Computer System," *Computer*, September 1990.

Cheriton, D., "The V Distributed System," *Communications of the ACM*, March 1988.

Demers, R. A., "Distributed Files for SAA," *IBM Systems Journal*, **27**, 3, 1988.

Gifford, D. K., Needham, R. M., Schroeder, M. D., "The Cedar File System," *Communications of the ACM*, March 1988.

Government Open Systems Interconnection Profile (GOSIP) Version 2, Federal Information Processing Standards Publication 146-1, National Institute of Standards and Technology, 1990.

IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Programming Interface (API) [C Language], IEEE Std 1003.1-1990, Institute of Electrical and Electronics Engineers Inc., 1990.

International Standard ISO 8571-1 – Information Processing Systems – Open Systems Interconnection – File Transfer, Access and Management, Reference Number:ISO 8571-1:1987(E), ISO TC97/SC21 Secretariat, ANSI, New York, New York, 1987.

Jalics, P. J., McIntyre, D. R., "Caching and Other Disk Access Avoidance Techniques on Personal Computers," *Communications of the ACM*, February 1989.

Liskov, B., "Distributed Programming in Argus," *Communications of the ACM*, March 1988.

Mooney, J. D., "Strategies for Supporting Application Portability," *Computer*, November 1990.

Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., Smith, F. D., "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, March 1986.

Mullender, S. L., van Rossum, G., Tanenbaum, A. S., van Renesse, R., van Saveren, H., "Amoeba: A Distributed Operating System for the 1990s," *Computer*, May 1990.

Network Programming, Sun Microsystems, Part Number:800-3850-10, Revision A, March 27, 1990.

"Next-generation NOSs Already Are Yielding Practical Benefits," *Data Communications*, May 1988.

Notkin, D., Black, A. P., Lazowska, E. D., Levy, H. M., Sanislo, J., Zahorjan, J., "Interconnecting Heterogeneous Computer Systems," *Communications of the ACM*, March 1988.

Notkin, D., Hutchinson, N., Sanislo, J., Schwartz, M., "Heterogeneous Computing Environment: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity," *Communications of the ACM*, February 1987.

"OSF Distributed Computing Environment Rationale," Open Software Foundation, May 14, 1990.

Ousterhout, J. K., et al., "The Sprite Network Operating System," *Computer*, February 1988.

POSIX: Portable Operating System Interface for Computer Environments, Federal Information Processing Standards Publication 151-1, National Institute of Standards and Technology, 1989.

Rashid, R. F., "Threads of a New System," *Unix Review*, August 1986.

Rifkin, A. P., Forbes, M. P., Hamilton, R. L., Sabrio, M., Shah, S., Yueh, K., "RFS Architectural Overview," *Summer Usenix Conference Proceedings*, Atlanta 1986.

Rose, M. T., **The Open Book**, Prentice-Hall Inc., 1990.

Roux, E., "OSI's Final Frontier: The Application Layer," *Data Communications*, January 1988.

Sandburg, R., et al., "Design and Implementation of the Sun Network File System," *Proceedings of the Summer Usenix Conference*, Portland, 1985.

Satyanarayanan, M., "A Survey of Distributed File Systems," *Annual Review of Computer Science*, Volume 4, 1989.

Satyanarayanan, M., "Scalable, Secure, and Highley Available Distributed File Access," *Computer*, May, 1990.

Saur, C.H., et al., "RT PC Distributed Services Overview," *ACM Operating Systems Review* **21**, 3, July 1987.

Spanier, S., "Comparing Distributed File Systems," *Data Communications*, December 1987.

Spector, A. Z., Kazar, M. L., "Uniting File Systems," *Unix Review*, March 1989.

Svobodova, L., "File Servers for Network-based Distributed Systems," *ACM Computing Surveys*, December 1984.

System V Interface Definition, Third Edition, Volumes 1–4, AT&T, 1989.

Tannenbaum, A. S., **Computer Networks**, Prentice-Hall Inc., 1981.

Verity, J. W., et al., "Rethinking the Computer," *Business Week*, November 26, 1990.

Walker, B. J., "Distributed UNIX Transparency: Goals, Benefits and the TCF Example," *CommUNIXations*, July/August 1989.

White, D., et al., "NOSs for Corporate Nets: How Do They Measure Up?", *Data Communications*, June 21, 1990.

Appendix B

Source Programs for Demonstrations

```
/*********************************************************************/
/*                            getlock.c                              */
/*                                                                   */
/*  Use F_GETLK with fcntl() to get the process ID of the process  */
/*  holding the lock for the file which is passed as an argument   */
/*********************************************************************/

#include <stdio.h>
#include <fcntl.h>

#define ERROR(s)   {fprintf(stderr,"%d-",errno); perror(s); exit(1);}

struct flock flstruc;
extern int errno;

main(argc,argv)
int argc;
char *argv[];
{
        int fd;

        /* Open the file for reading and writing */
        if ((fd = open(argv[1], O_RDWR)) < 0)
                ERROR("open");

        /* Set fields of the lock description argument */
        flstruc.l_type = F_RDLCK;
        flstruc.l_whence = 0;
```

```
        flstruc.l_start = 0;
        flstruc.l_len = 0;

        /* Use F_GETLK to get process ID of the process holding the lock */
        if(fcntl(fd, F_GETLK, &flstruc) < 0)
                ERROR("getlk");

        printf ("l_pid is %d\n", flstruc.l_pid);
}
```

```
/*************************************************************/
/*                    read_file.c                            */
/*                                                           */
/*  Read_file opens a file which is passed as a parameter,   */
/*  reads 2000 bytes a bytes at a time, pauses until a       */
/*  newline is received, and then continues to read data     */
/*  until end of file or a read error occurs.                */
/*************************************************************/

#include <stdio.h>

#define ERROR(s)   {fprintf(stderr,"%d-",errno); perror(s); exit(1);}
#define LC_NOTE {fprintf(stderr,  \
    "\nNOTE: - last-close demo may not behave as described\n");}

extern int errno;

main(argc,argv)
int argc;
char *argv[];
{
        int fd, retval, utflag, count=0;
        char c;

        /* update the access and modification times of the file to be */
        /* read to invalidate the cache for the file */
        if((utflag = utimes(argv[1],0)) < 0) perror("utimes");

        fd = open(argv[1], 0);
        if(fd < 0)
                ERROR("open");

        /* Read file one byte at a time */
        while(retval=read(fd, &c, 1)) {
                if(retval < 0)  {
                        fprintf(stderr,"read %d bytes\n", count);
                        ERROR("read");
                        }
                putchar(c);
                count++;
                if(count == 2000) {
                        if(utflag < 0) LC_NOTE;
```

```
                        retval = getchar();
}
                }
        fprintf(stdout,"read %d bytes\n", count);
}
```

```
/*********************************************************/
/*                    read_msg.c                         */
/*                                                       */
/*  Read a message from another process via fileX.       */
/*  Assume that fileX already exists and has mode 666.   */
/*********************************************************/

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define ERROR(s) {fprintf(stderr,"%d-",errno); perror(s); exit(1);}
extern int errno;
struct stat buf;

main ()
{
        int fd_msg, retval, i=0;
        char c, buffer[100];

        /* Exit if stat error, or wait until fileX is not length 0 */
        while (1)  {
                if ((stat ("fileX", &buf)) == -1)
                        ERROR("stat");
                if (buf.st_size != 0)
                        break;
                }

        /* Try to open fileX for reading */
        while((fd_msg = open ("fileX", O_RDWR)) == -1);

        /* Read the message from fileX */
        while (read(fd_msg,&buffer[i++],1) > 0);
        buffer[i] = '\0';

        /* Write message to stdout */
        write(1,buffer,strlen(buffer));
        close(fd_msg);

        /* Truncate the file to indicate to the writing process that the
```

```
        message was read */
    if ((fd_msg = open ("fileX", O_RDWR|O_TRUNC)) == -1)
            ERROR("open");
    close(fd_msg);
}
```

```
/*************************************************************/
/*                      read_n_changeID                      */
/*                                                           */
/* read_n_changeID opens a file (passed as the first argument) */
/* and reads 2000 bytes.  If the second argument is "user",    */
/* the real and effective user IDs are changed to the value    */
/* of the third argument.  If the second argument is "group",  */
/* the real and effective group IDs are changed to the value   */
/* of the third argument.  The rest of the file is read until  */
/* end of file or a read error occurs.                         */
/*************************************************************/

#include <stdio.h>
#include <sys/types.h>

#define ERROR(s)  {fprintf(stderr,"%x-",errno); perror(s); exit(1);}

#define USAGE   {fprintf(stderr, "usage: %s <filename> [group|user] <ID>\n",\
                   argv[0]); exit (1);}

extern int errno;

main (argc,argv)
int argc;
char *argv[];
{
    int fd, retval, count=0;
    char c;

    if (argc != 4)
        USAGE;
    if ((strcmp("user", argv[2])) && (strcmp("group", argv[2])))
        USAGE;

    /* Open the file for reading */
    if ((fd = open(argv[1], 0)) < 0)
        ERROR("open");

    /* read the file a byte at a time */
    while (retval=read(fd, &c, 1))   {
        if(retval < 0)  {     /* access was denied */
            fprintf(stdout,"read %d bytes\n", count);
```

```
            ERROR("read");
            }
        putchar(c);
        count++;
        if(count == 2000)  {
            if (!strcmp("user", argv[2]))  {  /* set the user ID */
                if ((retval = setuid(atoi(argv[3]))) < 0)
                    ERROR("setuid");
                }
            else if (!strcmp("group", argv[2]))  {  /* set group ID */
                if ((retval = setgid(atoi(argv[3]))) < 0)
                    ERROR("setgid");
                /* set effective uid = real uid so the effective
                   uid is no longer that of the super-user */
                if ((retval = seteuid(getuid())) < 0)
                    ERROR("seteuid");
                }
            }
        }
    fprintf(stdout,"read %d bytes\n", count);
}
```

```
/*********************************************************/
/*                      read_n_exec                      */
/*                                                       */
/*  Open a file (passed as the first argument) and read */
/*  2000 bytes.  Execute a file (passed as the second   */
/*  argument).                                          */
/*********************************************************/

#include <stdio.h>
#include <sys/types.h>

#define ERROR(s)   {fprintf(stderr,"%x-",errno); perror(s); exit(1);}

extern int errno;

main (argc,argv)
int argc;
char *argv[];
{

        int fd, retval, count=0;
        char c;

if (argc!=3)   {
                fprintf(stderr,"usage: %s <filename> <filename>\n",argv[0]);
                exit(1);
                }

        /* Open file for reading */
        if ((fd = open(argv[1], 0)) < 0)
                ERROR("open");

        /* Read 2000 bytes, a byte at a time */
        while (retval = read(fd, &c, 1)) {
                if(retval < 0)  {        /* access was denied */
                        fprintf(stderr,"read %d bytes\n", count);
                        ERROR("read");
                        }
                count++;
                putchar(c);
                if(count == 2000)
                        break;
```

```
        }

        /* Execute the file passed as the second argument */
        execl (argv[2], argv[2], NULL);
        printf ("%x-", errno);
        fflush(stdout);
        perror("execl");
        exit(1);
}
```

```
/*****************************************************/
/*                      setlock.c                    */
/*                                                   */
/*  Set an exclusive lock on the entire file which   */
/*  is passed as an argument, wait for a character   */
/*  of input, and then unlock the file.              */
/*****************************************************/

#include <stdio.h>
#include <fcntl.h>

#define ERROR(s)   {fprintf(stderr,"%d-",errno); perror(s); exit(1);}

struct flock flstruc;
extern int errno;

main(argc,argv)
int argc;
char *argv[];
{
        int fd, retval;

        /* Open file for reading and writing */
        fd = open(argv[1], O_RDWR);
        if(fd < 0)
                ERROR("open");

        /* Set fields of the lock description argument */
        flstruc.l_type = F_WRLCK;
        flstruc.l_whence = 0;
        flstruc.l_start = 0;
        flstruc.l_len = 0;

        /* Set an exclusive lock on the file */
        if(fcntl(fd, F_SETLK, &flstruc) < 0)
                ERROR("wrlck");
        printf("%s locked\n",argv[1]);

        /* Wait for a character of input */
        retval = getchar();

        /* Unlock the file */
```

```
        flstruc.l_type = F_UNLCK;
        if(fcntl(fd, F_SETLK, &flstruc) < 0)
                ERROR("unlck");
        printf ("%s unlocked\n",argv[1]);
}
```

```
/*********************************************************/
/*                       set_user_ID                     */
/*                                                       */
/* This program, which is executed by the program        */
/* read_n_exec,  has its set-user-ID bit on and continues */
/* to read the file specified by file descriptor 3.      */
/*********************************************************/

#include <stdio.h>
#include <sys/types.h>

#define ERROR(s)   {fprintf(stderr,"%x-",errno); perror(s); exit(1);}

extern int errno;

main (argc,argv)
int argc;
char *argv[];
{
        int  retval, count=2000;
        char c;

        /* Continue to read the file a byte at a time */
               while (retval = read(3, &c, 1)) {
                if (retval < 0)    {   /* access was denied */
                        fprintf(stdout,"read %d bytes\n", count);
                        ERROR("read");
                        }
                putchar(c);
                count++;
                }
        fprintf(stdout,"read %d bytes\n", count);
}
```

```
/*****************************************************************/
/*                      show_IDs                                 */
/*                                                               */
/*  This program, which is a modification of read_n_changeID,    */
/*  displays the effective user ID and group ID stored in the    */
/*  process table and stored in the file descriptor table entry. */
/*****************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <sys/types.h>

#include <kvm.h>
#include <fcntl.h>

#include <sys/param.h>
#include <sys/time.h>
#include <sys/proc.h>
#define KERNEL     /* kludge needed to include the file structure */
#include <sys/file.h>

#define ERROR(s)  {fprintf(stderr,"%x-",errno); perror(s); exit(1);}

#define USAGE   {fprintf(stderr, "usage: %s <filename> [group|user] <ID>\n",\
argv[0]); exit (1);}

extern int errno;

main (argc,argv)
int argc;
char *argv[];
{
    int fd, retval, count=0;
    char c;
    struct stat buf;

    if (argc != 4)
        USAGE;
    if ((strcmp("user", argv[2])) && (strcmp("group", argv[2])))
        USAGE;
```

```
    /* Open the file for reading */
    if ((fd = open(argv[1], 0)) < 0)
        ERROR("open");

    kernel_info("Before setuid()/setgid()", argv[1]);

    /* read the file a byte at a time */
    while (retval=read(fd, &c, 1))   {
        if(retval < 0)  {     /* access was denied */
            fprintf(stdout,"read %d bytes\n", count);
            ERROR("read");
            }
        count++;
        if(count == 2000)  {
            if (!strcmp("user", argv[2]))  {  /* set the user ID */
                if ((retval = setuid(atoi(argv[3]))) < 0)
                    ERROR("setuid");
                printf ("\nChanged the real & effective user IDs to %s\n", argv[3]);
                }
            else if (!strcmp("group", argv[2]))  {  /* set group ID */
                if ((retval = setgid(atoi(argv[3]))) < 0)
                    ERROR("setgid");
                printf ("Changed the real and effective user GIDs to %s\n", argv[3]);
                /* set effective uid = real uid so the effective
                   uid is no longer that of the super-user */
                if ((retval = seteuid(getuid())) < 0)
                    ERROR("seteuid");
                printf ("Set the effective uid=real uid so the effective\n");
                printf ("uid is no longer that of the super-user\n");
                }
            }
        }
   kernel_info("After setuid()/setguid()", argv[1]);
   fprintf(stdout,"\nread %d bytes\n", count);
}


kernel_info(msg, filename)
/*********************************************************/
/* Display information retrieved from the process table */
/* and the file descriptor table.                        */
/*********************************************************/
```

```
char *msg, *filename;
{
kvm_t *kd;
struct proc *procstruct;
struct user *userstruct;
struct ucred u_cred, u_cred_addr;
struct file file_addr;
int fd, retval;

    printf ("\n%s:\n", msg);
    /* Get a pointer to a kernel identifier so that the kernel can
        be examined */
    kd = kvm_open(NULL,NULL,NULL, O_RDONLY,"kernel_info");

    /* Get the u-area for this process */
    procstruct = kvm_getproc(kd,getpid());
    if ((userstruct = kvm_getu(kd,procstruct)) == NULL)
        ERROR("kvm_getu");

    /* Copy data from the kernel image for process credentials */
    if ((retval=kvm_read(kd, procstruct->p_cred, &u_cred,
         sizeof(struct ucred))) == -1)
        ERROR("kvm_read");

    printf ("\n    Information from the process table:\n");
    printf ("        The effective uid is %d\n", u_cred.cr_uid);
    printf ("        The effective gid is %d\n", u_cred.cr_gid);
    /*
    printf ("        The real uid is %d\n", u_cred.cr_ruid);
    printf ("        The real gid is %d\n", u_cred.cr_rgid);
    */

    /* Copy data from the kernel image for file structures for open files */
    retval = kvm_read(kd, userstruct->u_ofile_arr[3], &file_addr,
                                                    sizeof(struct file));
    if (retval == -1)
        ERROR("kvm_read");

    /* Copy data from the kernel image for the credentials of the user who
       opened the file */
    retval = kvm_read(kd, file_addr.f_cred, &u_cred_addr,
                                                sizeof(struct ucred));
```

```
    if (retval == -1)
            ERROR("kvm_read");

    printf ("\n    Information from the file descriptor table:\n");
    printf ("        The effective uid of the user who opened %s is %d\n",
                                            filename, u_cred_addr.cr_uid);
    printf ("        The effective gid of the user who opened %s is %d\n",
                                            filename, u_cred_addr.cr_gid);
    /*
    printf ("        The real uid of the user who opened %s is %d\n",
                                           filename, u_cred_addr.cr_ruid);
    printf ("        The real gid of the user who opened %s is %d\",
                                           filename, u_cred_addr.cr_rgid);
    */

    kvm_close(kd);
}
```

```
/*************************************************************/
/*                      write_msg.c                          */
/*                                                           */
/* Pass a message to another process through the file fileX. */
/* Assume that fileX exists and has mode 666.                */
/*************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

#define ERROR(s)    {fprintf(stderr,"%d-",errno); perror(s); exit(1);}

extern int errno;
struct stat buf;

main ()
{
        int fd_msg;
        char ch;

        /* Exit if stat error, or wait until file size is 0 */
        while (1) {
                if ((stat ("fileX", &buf)) == -1)
                        ERROR("stat");
                if (buf.st_size == 0)
                        break;
                }

        /* Open fileX and then chmod 0 to prevent the process
           read_msg from opening fileX */
        if ((fd_msg = open ("fileX", O_RDWR)) == -1)
                ERROR("open");
        chmod ("fileX", 0);

        /* Read the message from stdin and write the message to fileX */
        while ((ch = getc(stdin)) != EOF)
                if (write (fd_msg, &ch, 1) < 0)
                        ERROR("write");
```

```
        /* Restore the permission bits so that the process read_msg
           can open fileX */
        chmod ("fileX", 0666);
        close(fd_msg);
}
```

```
/*************************************************************************/
/*                          WriteStdIO.c                                 */
/*                                                                       */
/*  Open terminal for reading raw input.  Open the file passed as an     */
/*  argument for writing output using stdio. Read and write data, a      */
/*  byte at a time, until a ^D is recieved.  Print the number of         */
/*  bytes that were read and written.                                    */
/*************************************************************************/

#include <stdio.h>
#include <fcntl.h>

#define ERROR(s)   {system("stty -raw"); fprintf(stderr,"%d-",errno); \
                    perror(s); exit(1);}

extern int errno;

main(argc,argv)
int argc;
char *argv[];
{
        int fd, retval, count = 0;
        FILE *ostream;
        char c;

        /* Open terminal for reading raw input */
        system("stty raw");
        if ((fd = open("/dev/tty", O_RDONLY)) < 0)
                ERROR("open input");
        ostream = fopen(argv[1], "w");
        if(ostream == NULL)
                ERROR("fopen output");
        ostream->_bufsiz = 10;

        while(retval=read(fd, &c, 1))  {  /* Read & write data until ^D */
                if(retval < 0){
                        fprintf(stdout,"read %d bytes\n", count);
                        ERROR("read");
                }
                if(c == 4) /* ^D */ break;
                        count++;
                retval = putc(c, ostream);
```

```
        if(retval == EOF){
                fprintf(stdout,"read %d bytes\n",count);
                ERROR("write");
          }
    }
    system("stty -raw");
    fprintf(stdout,"read/wrote %d bytes\n", count);
}
```

```
/*********************************************************************/
/*                          WriteUnixIO.c                            */
/*                                                                   */
/*  Open terminal for reading raw input.  Open the file passed as an */
/*  argument for writing raw output. Read and write data, a byte at  */
/*  a time, until a ^D is recieved.  Print the number of bytes that  */
/*  were read  and  written.                                         */
/*********************************************************************/

#include <stdio.h>
#include <fcntl.h>
extern int errno;

#define ERROR(s)   {system("stty -raw"); fprintf(stderr,"%d-",errno); \
                    perror(s); exit(1);}

main(argc,argv)
int argc;
char *argv[];
{
        int fd, fdo, retval, count=0;
        char c;

        system("stty raw");
        fd = open("/dev/tty", O_RDONLY);
        if(fd < 0)
                ERROR("open input");

        fdo = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 00666);
        if(fdo < 0)
                ERROR("open output");

        while( retval=read(fd, &c, 1) ) {  /* Read & write data until ^D */
                if(retval < 0) {
                        fprintf(stdout,"read %d bytes\n", count);
                        ERROR("read");
                        }
                if(c == 4) /* ^D */ break;
                count++;
                retval=write( fdo, &c, 1);
                if(retval < 0) {
                        fprintf(stdout,"read %d bytes\n",count);
```

```
                        ERROR("write");
                        }
                }
        system("stty -raw");
        fprintf(stderr,"read/wrote %d bytes\n", count);
}
```

NIST-114A
(REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NIST/SP-500/186

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
April 1991

4. TITLE AND SUBTITLE

Issues in Transparent File Access

5. AUTHOR(S)

Karen Olsen and John Barkley

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED
Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

Same as item #6

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

For a computer system attached to a network, the network provides connectivity to many other systems whose file systems may be very different from the local file system. However, there is no standard way for an application to "transparently" access files on several file systems whose access characteristics may differ from the access characteristics of the local file system. Transparent file access means that remote files are accessed as though they were local. The Institute of Electrical and Electronics Engineers (IEEE) 1003.8 Transparent File Access (TFA) Working Group of the POSIX Standards Committee (IEEE P1003) has undertaken the development of an application programming interface specification based on the IEEE 1003.1-1990 Standard. The objective of IEEE 1003.8 is to permit access to the widest possible range of file systems which can resemble the file system of IEEE 1003.1-1990. This report presents the major issues and problems whose resolution form the basis of the IEEE 1003.8 TFA Standard. Some issues are illustrated with examples and demonstrations using NFS, the most widely used implementation for accessing remote files on a network.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

file server; file system; network; operating system; POSIX; UNIX

13. AVAILABILITY

X	UNLIMITED
	FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
X	ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
X	ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
86

15. PRICE

ELECTRONIC FORM

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SYSTEMS TECHNOLOGY

Superintendent of Documents
Government Printing Office
Washington, DC 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Institute of Standards and Technology Special Publication 500-.

Name _____

Company _____

Address _____

City _____ State _____ Zip Code _____

(Notification key N-503)

☆ U.S. Government Printing Office : 1991 - 281-618/42731

NIST Technical Publications

Periodical

Journal of Research of the National Institute of Standards and Technology—Reports NIST research and development in those disciplines of the physical and engineering sciences in which the Institute is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Institute's technical and scientific programs. Issued six times a year.

Nonperiodicals

Monographs—Major contributions to the technical literature on various subjects related to the Institute's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NIST, NIST annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NIST under the authority of the National Standard Data Act (Public Law 90-396). NOTE: The Journal of Physical and Chemical Reference Data (JPCRD) is published quarterly for NIST by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements are available from ACS, 1155 Sixteenth St., NW., Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Institute on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NIST under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NIST administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NIST research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NIST publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NIST publications—FIPS and NISTIRs—from the National Technical Information Service, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NIST pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NIST Interagency Reports (NISTIR)—A special series of interim or final reports on work performed by NIST for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service, Springfield, VA 22161, in paper copy or microfiche form.

U.S. Department of Commerce
National Institute of Standards and Technology
Gaithersburg, MD 20899